Improving Visualisation of Large Multi-Variate Datasets: New Hardware-Based Compression Algorithms and Rendering Techniques

---

A thesis submitted in fulfilment of the
requirements for the Degree of
Master of Science in Computer Science
at the
University of Canterbury
by
Alexander Chernoglazov

University of Canterbury
2012

Abstract

Spectral computed tomography (CT) is a novel medical imaging technique that involves simultaneously counting photons at several energy levels of the x-ray spectrum to obtain a single multi-variate dataset. Visualisation of such data poses significant challenges due its extremely large size and the need for interactive performance for scientific and medical end-users. This thesis explores the properties of spectral CT datasets and presents two algorithms for GPU-accelerated real-time rendering from compressed spectral CT data formats. In addition, we describe an optimised implementation of a volume raycasting algorithm on modern GPU hardware, tailored to the visualisation of spectral CT data.

Table of Contents

List of Figures

List of Tables

Acknowledgments

Dr. R. Mukundan, Computer Science and Software Engineering, University of Canterbury
Dr. Raphael Grasset, HITLabNZ
Dr. Anthony Butler, Director of Bioengineering, University of Otago, Christchurch
Dr. Tim Bell, Computer Science and Software Engineering, University of Canterbury
Niels de Ruiter, HITLabNZ
The MARS project team
Everyone at the HITLab
Family and friends

Chapter I

Introduction

This project aims to improve the visualisation of large multi-variate datasets generated by a novel medical imaging technique: Spectral Computed Tomography (spectral CT). High quality visualisation is essential for further progress in the field of spectral CT imaging; obtaining an efficient and interactive visualisation, however, is largely dependent of the size of the dataset and remains highly challenging for spectral CT data. The goal of this project is to investigate the possibility of rendering directly from compressed data formats in order to enable the study of large spectral CT datasets at their native resolution.

Spectral CT is a new technique that has the potential to vastly improve on the established standard CT imaging technology. Essentially, spectral CT measures the attenuation of x-rays over several energy ranges of the electromagnetic spectrum separately, allowing for what is known as spectral, spectroscopic, or true colour imaging [1]. A comparison of current CT technologies is given in Fig. 1.1.

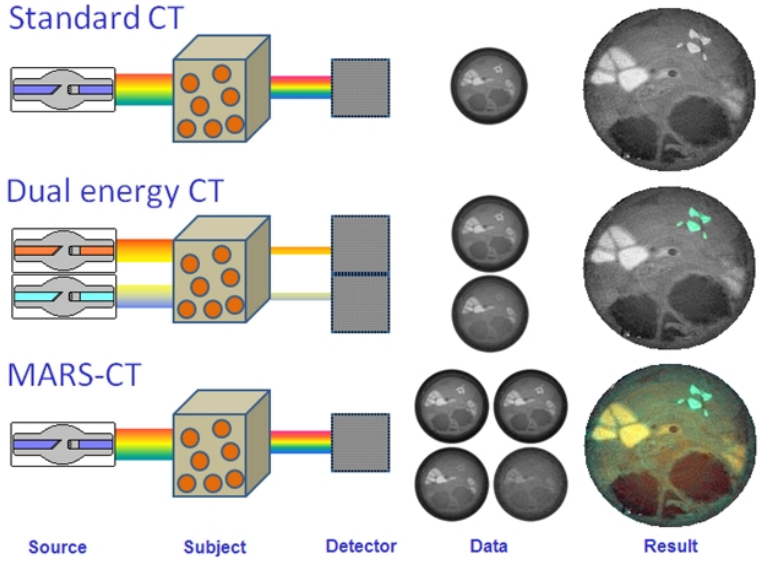

Figure 1.1: Comparison of current CT technologies with MARS-CT.

After a spectral CT scanner acquires raw projection images, they are pre-processed and reconstructed (examples of reconstructed slices are given in Fig. 1.3), creating a volumetric dataset that possesses certain unique properties. These *spectral CT datasets* can be visualised using custom data intermixing algorithms combined with standard volume rendering techniques.

It is expected that the medical benefits of spectral CT will range from easier detection of cancerous lesions [2] to enhanced diagnosis of atherosclerosis and non-alcoholic fatty liver disease [3]. These benefits, along with the principles of spectral CT technology, are explained in section 2.1.

This work is conducted in the context of the Medipix All Resolution System (MARS) project [4]. The MARS project is based in the University of Canterbury, and, along with a number of domestic and international partners, aims to develop the world's first spectral CT scanner (referred to as the MARS-CT scanner or the MARS-CT system). Development is divided into different tasks, including physically designing and building the scanner, creating the necessary image acquisition and data processing software and studying visualisation of datasets produced by the scanner. Clinical and scientific applications of the new system are also being explored. An overview of the MARS-CT scanner and spectral CT technology is presented in section 2.1.

The MARS project requires its visualisation software to be able to load and display large multi-gigabyte datasets (see section 3.1.1 for further discussion of the size of spectral CT datasets) at interactive frame rates on commodity graphics hardware . Currently, visualisation is performed on consumer and workstation-grade graphics processing units (GPUs) and is limited by the amount of video memory present on the device that is being used for rendering. Downsampling the dataset or only viewing a part of it are the only options if it cannot fit into available GPU memory. Such inability to view datasets at full resolution limits the effectiveness and usage of the entire system.

It would be highly beneficial for the MARS project to be able to explore and study spectral CT datasets at their native resolution. A potential approach to overcome the limited amounts of graphics memory on current GPUs is to compress spectral CT data before visualisation and then render directly from a compressed format.

1.1 Research Objectives

The objective of this project is to reduce the size of a large spectral CT dataset using compression whilst being able to display the data at interactive frame rates

on GPU hardware using volume rendering techniques. Fig. 1.2 shows a moderately large spectral CT dataset visualised by currently available methods. This representative dataset has been downsampled to 25% of its original size (from 1024x1024x512 voxels to 512x512x512 voxels) in order to be displayed. This project aims to increase the size of datasets that can be loaded and studied. In order to achieve this, several tasks need to be completed.

First of all, compression methods for spectral CT data that allow for random access, preserve sufficient detail to be relevant for scientific use (and possibly clinical applications, although this is not a priority at this stage of development of the MARS-CT scanner) and allow decompression and visualisation at interactive frame rates need to be developed.

At present, the characteristics of spectral CT data remain poorly explored and compression has not been studied. This thesis investigates which properties of spectral CT data could be used to create specialised compression schemes and also explores other methods of compression that do not take advantage of special properties of spectral CT datasets.

Real-time rendering from a compressed spectral CT dataset on GPU hardware using modern programming tools and languages also needs to be examined. Numerous 3D volume rendering algorithms have been successfully implemented on

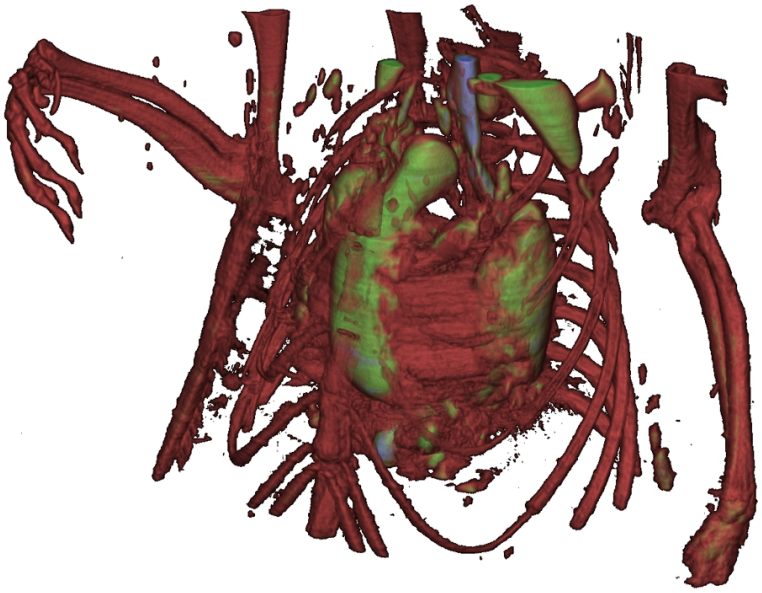

Figure 1.2: A 512^3 4-energy bin spectral CT dataset (1GB in total), showing the chest area of a mouse, as visualised by currently available software (MARSCTExplorer). Image courtesy of Niels de Ruiter.

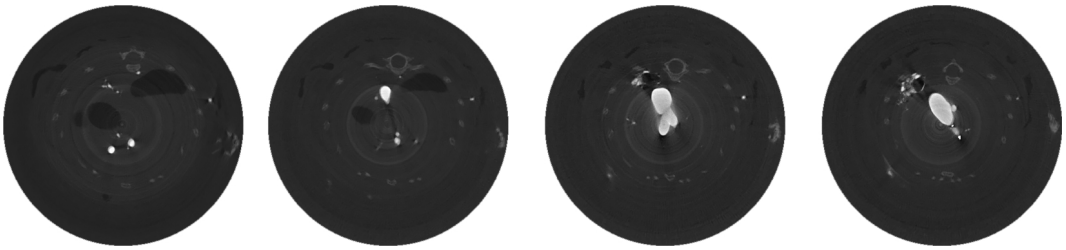

Figure 1.3: 4 slices of the Mouse12 dataset that were acquired by the MARS-CT scanner and reconstructed by the MARS project team. These slices are stacked to create a large volumetric dataset that can be rendered using volume raycasting methods.

GPUs, but the use of real-time decompression of volume data as part of a volume rendering application is uncommon. This problem is highly specific and limits the range of techniques that can be applied to solve it.

Finally, visualisation of spectral CT data needs be interactive to help medical and scientific users easily explore and browse the dataset. This requires all algorithms executed during rendering to be well-optimised. Previous research indicates that the use of compression inevitably results in a decrease in performance, so acceleration techniques, in particular spatial data structures, need to be investigated.

In summary, this project aims to explore four main research directions:

- Compression techniques that reduce spectral CT datasets of arbitrary size into formats suitable for fast decompression on GPU hardware.

- Algorithms able to rapidly decompress spectral CT data on GPU hardware and render a 3D visualisation at interactive frame rates.

- Algorithmic and platform-specific optimisations that may improve the execution speed and/or visual quality of GPU-based compression and rendering algorithms for spectral CT data.

- Evaluation of the quality of spectral CT datasets compressed using lossy methods.

1.2 Contribution

This thesis continues research into the properties of spectral CT datasets and provides a first look at the potential techniques that may be used to compress

spectral CT data.

In addition, this research presents a new software application, MARSCUDAVR, designed for the purpose of visualising compressed spectral CT datasets on GPU hardware. Methods of optimising volume raycasting algorithms to achieve real-time frame rates while rendering multi-gigabyte spectral CT datasets are also discussed.

1.3 Thesis Outline

Chapter 2 describes spectral CT technology in detail, along with the various visualisation techniques for CT and other volumetric data. Image and texture compression algorithms as applied to real-time rendering are examined and the role that GPGPU (General Purpose Computing on Graphics Processing Units), and specifically NVIDIA's CUDA programming language plays in this research is explained.

Chapter 3 contains the first half of this research, covering the properties of spectral CT datasets and presenting two compression algorithms that support real-time decompression of spectral CT data on GPU hardware. Both algorithms have been designed to allow for easy integration into volume rendering applications.

Chapter 4 describes MARSCUDAVR, the application created over the course of this research for the purpose of interactively visualising large compressed spectral CT datasets. Practical implementation of compression, decompression and visualisation algorithms on modern GPU hardware is examined, along with some methods for improving their execution speed and visual quality.

Chapter 5 is concerned with evaluating the quality of images generated by the new compression and visualisation algorithms and comparing it to the quality provided by the methods currently used for visualising spectral CT data. The speed of rendering algorithms is also compared and the effect of underlying hardware architectures on performance is discussed.

Chapter 6 summarises this research project and discusses its primary findings. It also explores directions for future research in spectral CT data visualisation by identifying areas where improvement is still required and suggesting possible solutions.

Appendices A, B and C contain additional information that the reader may find useful. In particular, Appendix C contains a glossary of terms that are frequently used throughout this thesis.

Chapter II

Related Work

Due to the nature of the context of this research, this chapter provides an overview of previous work related to this project, covering both the fields of medical imaging and computer science. First, spectral computed tomography (spectral CT) is covered in section 2.1 and the design and operation of the MARS-CT scanner is briefly explained. Next, traditional techniques for visualising 2D images, 3D volume data, medical and spectral CT datasets are examined in section 2.2.

Specific aspects of GPU programming related to this project are introduced in section 2.3, examining both the underlying concepts of GPGPU and dedicated GPGPU programming languages such as CUDA. Next, compression of images and textures for the purposes of real-time rendering from a compressed format is examined in section 2.4 and finally the applicability of lossy compression to medical images is explored in section 2.5.

2.1 Spectral CT Technology

Computed Tomography is a medical image acquisition technique developed in the 1970's and commonly attributed to Sir Godfrey Hounsfield [6]. The first paper on CT published by researchers from the University of Canterbury was in 1971, by Bates and Peters [7], and the first time a CT scanner was used in a clinical setting was in 1972, when a frontal lobe tumour was successfully detected using the new technology. For a history of CT and its clinical applications the reader can be referred to a review by Kalender [8].

A typical CT scanner takes a large number of *projection images* from different angles around a human subject or an object being examined. An example of one possible CT scanner design is given in Fig. 2.1. During a scan, x-rays are sent from a source to a detector and are attenuated (lose power) by passing through different materials such as soft tissues or bones. Their remaining intensity is measured by a detector, which generates a projection image per view angle.

An example of a projection image taken by a CT scanner is shown in Fig. 2.2. Projection images taken over the course of a scan are combined and reconstructed

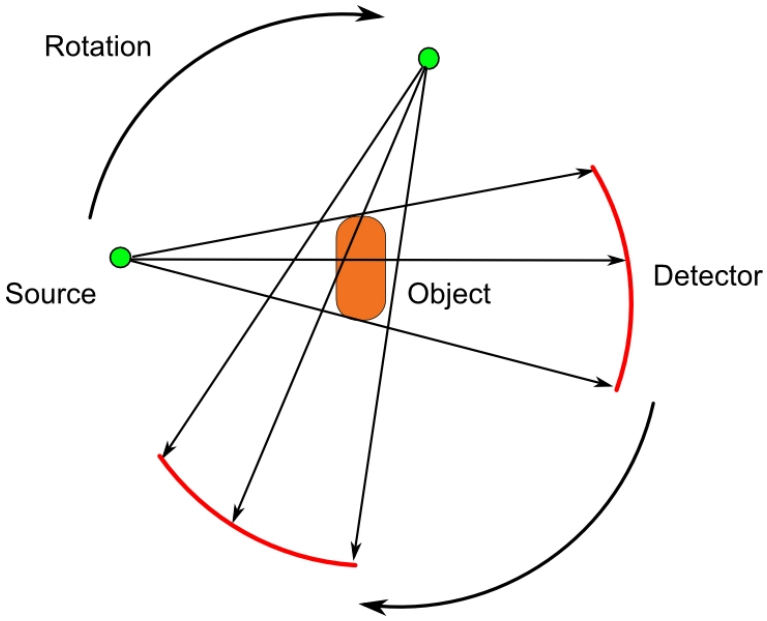

Figure 2.1: Design of a wide fan beam CT scanner [5]. The source and detector are fixed relative to each other and rotate about the object being scanned, taking a projection image at each orientation. Several other CT scanner types exist.

to form a volumetric dataset using a variety of techniques, such as back projection or algebraic reconstruction.

The difference between standard and spectral CT scanners lies not in mechanical design, but in the detectors used for measuring x-ray intensity. Spectral CT

Figure 2.2: Projection image of a thoracic (chest) section of a mouse. The front paws, rib cage and internal organs are visible.

detectors measure the attenuation of different energies of the x-ray spectrum separately, whereas standard CT only detects the total attenuation over the entire measurable spectrum. A comparison with colour photography can be drawn, where the channels (for example, red, green and blue) describe the same object or scene, but convey different information by separately measuring the intensity of light at different wavelengths of the visible spectrum. Spectral CT detectors perform the same measurement, but in the x-ray region of the electromagnetic spectrum, which ranges from 10 to 150 kiloelectronvolts (keV).

In spectral CT, the detector counts photons at several energy levels simultaneously, thereby creating a number of volumetric datasets from a single scan. Each one of these datasets, also referred to as *energy bins*, describes the attenuation of x-rays over a certain range of the x-ray spectrum, for example between 15 and 80 keV or between 35 and 80 keV. Taken together, energy bins form a spectral CT dataset, which is essentially a collection of 3D volumetric datasets that describes a physical object that is being scanned in slightly different ways.

Data contained in energy bins can be treated and studied as a set of independent datasets, or as a single 4D dataset (the fourth dimension being the x-ray frequency). The latter approach is the goal of the MARS project.

The main benefit of spectral CT is that it allows for much clearer diagnostic imaging of tissues, as photon attenuation through a material varies non-linearly with respect to energy [9]. In practice, this means that materials can be discrimi-

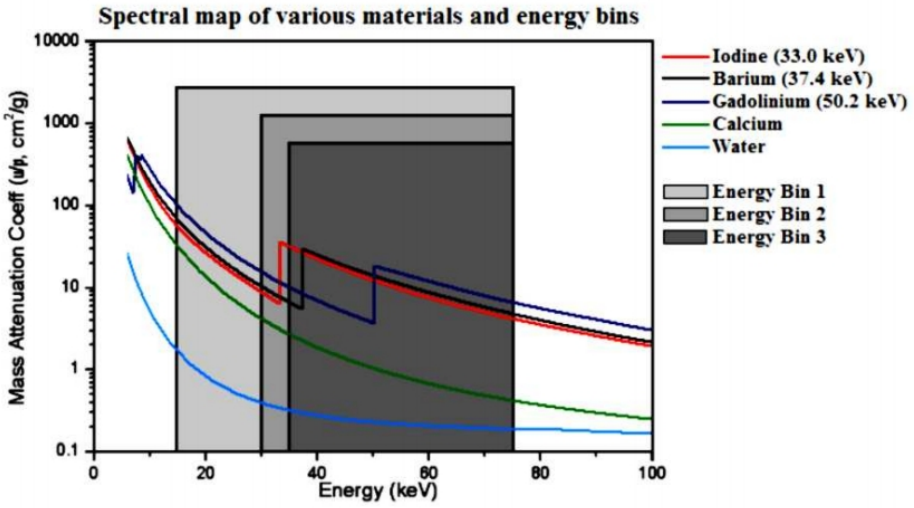

Figure 2.3: Material attenuation over a part of the x-ray spectrum. Responses of different materials vary based on x-ray energy levels. Image courtesy of the MARS project.

nated based on their responses to x-rays of different energies. For instance, iodine and barium are two commonly used contrast agents in radiology, but, as shown in Fig. 2.3, they attenuate photons very similarly over most of the x-ray spectrum. The only major difference is at their *k-edges*, located at 33.0 and 37.4 keV respectively. Spectral CT is able to differentiate between these materials, whereas standard CT cannot [10].

For example, as shown in Fig. 2.3, energy bin 2 measures the total attenuation from 30 to 75 keV. This bin includes iodine's k-edge, but energy bin 3, which measures the total attenuation over the 35-75 keV range, does not. Therefore, these bins can be algorithmically combined to show differences by, for example, subtracting one bin from another.

A spectral CT dataset is acquired with a single exposure, which removes the need for registration, which is a common problem when medical images are acquired at different times, from different viewpoints or by different sensors. For instance, single photon emission computed tomography (SPECT) can be combined with magnetic resonance imaging (MRI) to enhance detection of liver cancers. The two datasets, however, need to be aligned in order to receive the benefits of combining these imaging modalities [11]. Another advantage of spectral CT technology is that there is only a single source of Poisson noise [2], as opposed to dual-energy CT systems.

As noted in 1, it is expected that in the future spectral CT will be used to detect specific biological markers of diseases such as lung infections, atherosclerosis, liver disease and degenerative joint disease. Spectroscopic imaging will enhance diagnosis of various medical conditions by precisely identifying the chemical composition of tissues and will lead to better, more individualised treatment for patients.

2.1.1 MARS-CT Scanner

The MARS project, a collaboration between the University of Canterbury and a number of domestic and international partners, is in the process of building a spectral CT imaging system called the MARS-CT scanner or the MARS-CT system. Its current iteration is built for imaging of biological samples and small animals and is able to scan objects at most 100mm in diameter and 200mm in length [12]. The gantry, containing a detector and an x-ray source, is capable of 360° rotation around the target. The gantry is stopped at every image angle and a projection image is acquired. Projection images are then reconstructed using cone beam filtered back projection.

Currently, the MARS-CT-3 scanner takes around 2.5 seconds for a single rotation stop [13], although mechanical improvements aim to reduce the total imaging

time to less than five minutes. When operated in fast mode, the MARS-CT scanner is capable of scanning live animals, and in October 2011 the first successful scan of a sedated live mouse has been performed.

The MARS-CT scanner can use several detector chips, such Medipix2, Timepix or Medipix3, which have been developed by a collaboration involving over 20 universities lead by CERN (European Organisation for Nuclear Research). Medipix2 and Medipix3 chips consist of a matrix of 256x256 pixels, with each pixel occupying a total area of 55x55μm [14]. In detectors belonging to the Medipix series, incoming photons interact with a semiconductor sensor layer to produce a charge cloud. An ASIC (Application-Specific Integrated Circuit) that is bonded to the sensor layer analyses the charge cloud to determine the photon's energy. Each pixel uses an amplifier, a shaper and a discriminator to measure the input pulse generated by a photon. This is a form of *pulse height analysis* which detects the energy of an incoming photon and determines which energy bin it should belong to.

In case of the MARS-CT scanner, data will normally be acquired in "low threshold" mode, which means that energy bins will overlap. As an example, consider a scan that is performed with the highest measured energy being 80 keV and the lowest measured energy being 15 keV. The MARS-CT scanner is able to partition this range into several, user-defined energy bins, for example 15 keV - 80 keV, 23 keV - 80 keV, 35 keV - 80 keV and so on. The shorthand notation for these ranges is simply 15 keV, 23 keV and so on, where the lower bound is taken to be the name of the energy bin and the upper energy limit is implied, as it is the same for all energy bins of spectral CT dataset. This notation will be used throughout this thesis.

2.2 Visualisation of CT Datasets

Since the invention of CT technology in the 1970's, various methods of visualising CT datasets have been attempted, taking into account the hardware limitations of the time. For instance, real-time 3D visualisation was not possible on the hardware of the first decades following the development of the first CT scanners. However, modern desktop computers equipped with mid-range GPUs are able to display large datasets at interactive frame rates. This trend is expected to continue.

This section covers a range of techniques used for visualisation of CT datasets, from the basic 2D slice-by-slice method to advanced 3D visualisation tools providing real-time segmentation and near-photorealistic image quality.

The most common way to present a CT dataset in a form suitable for human

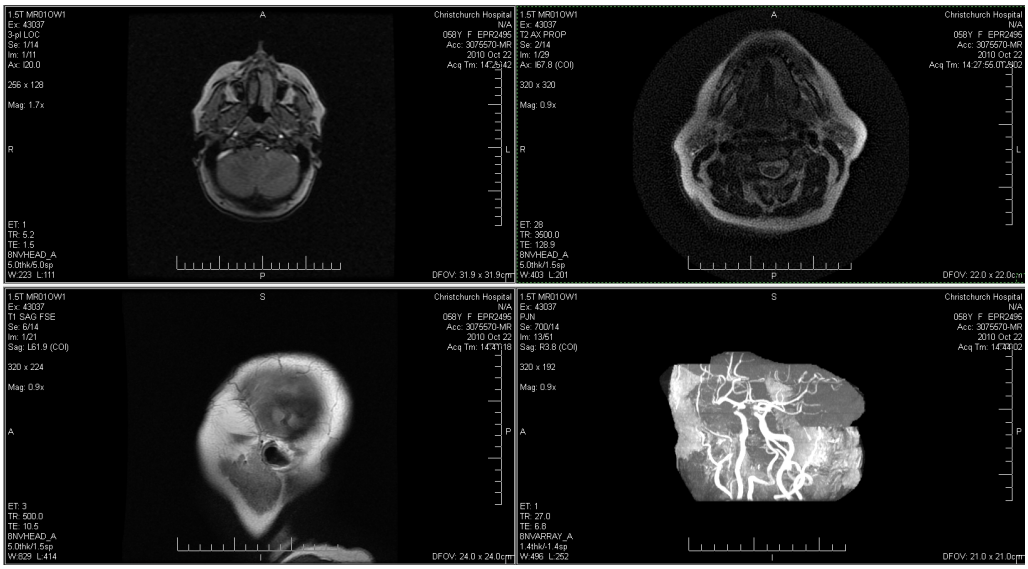

Figure 2.4: Visualisation of an MRI scan with eFilmLite [15]. Different 2D views are shown along with a basic 3D rendering.

interaction is by using the concept of *slices*: a set of cross-sections of an object being scanned. Slices are thereafter visualised either as independent 2D images (by browsing the slices) or integrally (as a 3D dataset by means of volume rendering or other similar techniques). Display of 2D images, even of large size, generally poses no problems on modern hardware, whereas 3D visualisation is significantly more complex yet potentially more powerful.

2.2.1 2D Visualisation

The oldest method of visualisation of medical CT datasets is slice-by-slice exploration. It is a simple technique that requires no specialised algorithms or powerful hardware and can be considered an extension of the standard (non-CT) x-ray image visualisation process. As such, it can be readily accepted by radiologists and does not require any familiarity with computer graphics or volume reconstruction. While this technique is fairly basic, it can nevertheless be extended by, for example, pre-processing of slices in order to segment tissues.

Standard 2D visualisation software will generally display several views simultaneously and include the option to show three standard orientations (axial, sagittal and coronal), synchronise views, apply ramp functions, increase contrast and so on. 2D visualisation is the most widely used technique in clinical practice, but it is currently being challenged by 3D visualisation, which offers substantial improvements. Most commercially available software (such as eFilmLite, Fig. 2.4) is able

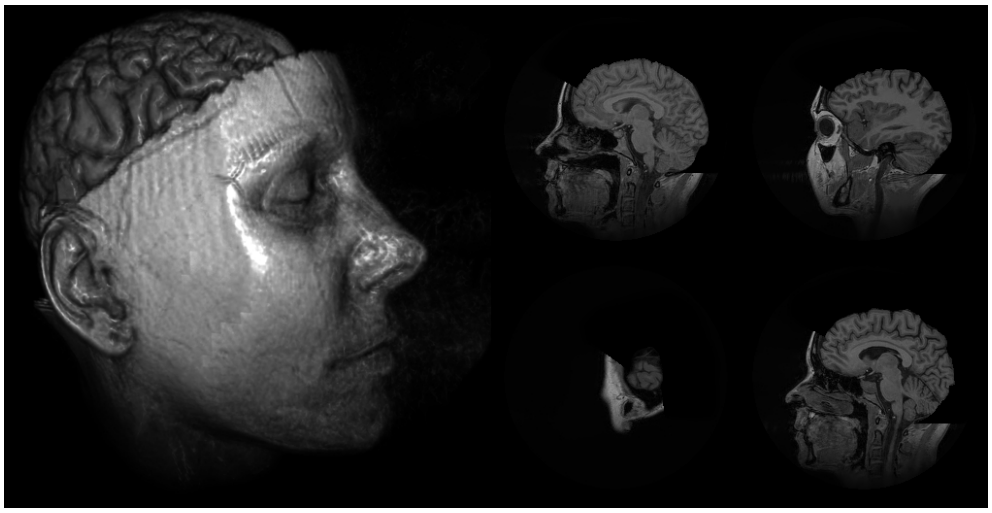

Figure 2.5: Left: a 3D render of the MRBrain dataset (99 slices of 256x256 pixels each). Right: examples of 2D slices that comprise the MRBrain dataset.

to visualise data in both 2D and 3D.

2.2.2 3D Visualisation

This type of visualisation involves displaying the entire 3D dataset, or part of it, as volume data. Images rendered using techniques such as volume raycasting can convey significantly more information than pure 2D visualisation. Regions of interest (such as an organ in the human body) normally span multiple slices in a CT dataset, but can be seen in their entirety in a 3D rendering. Depth cues, (for example, using volumetric lighting or shadows) can be added to the scene to further increase the level of detail perceived by the user of the system.

However, this type of visualisation introduces certain constraints and, in general, is more demanding of the hardware because a much larger portion of the dataset is processed using complicated algorithms and displayed at once. However, its primary advantage is that the user no longer needs to build up a mental image of a 3D object by viewing individual 2D slices. A comparison between visualising 2D slices and viewing the entire dataset in 3D is shown in Fig. 2.5.

3D visualisation may be used to present a large volumetric dataset to the user, which may lead to several possible issues, such as areas of interest being occluded by other parts of the volume or important features being skipped as empty space during rendering. These issues can be solved by focusing on UI design and specifically by letting the user adjust a number of parameters, such as opacity levels, thresholds and transfer functions.

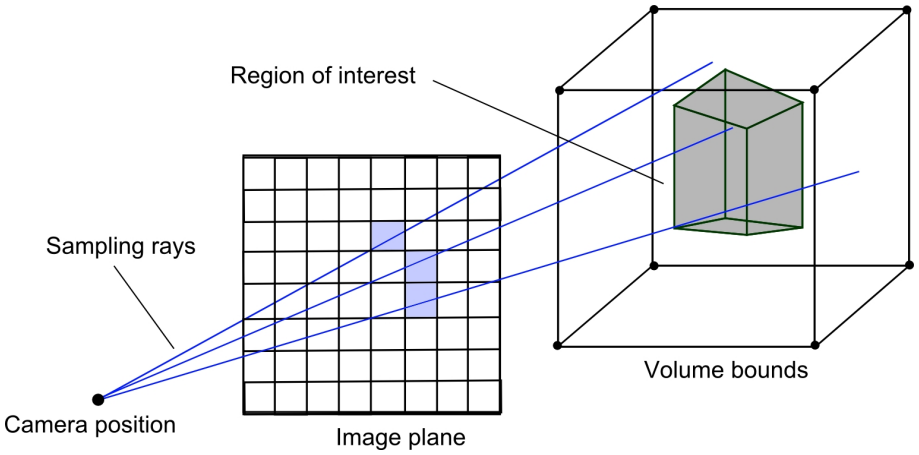

Figure 2.6: Basic volume raycasting. A camera sends rays through an image plane, with rays sampling inside the volumetric dataset and generating an image pixel-by-pixel.

2.2.2.1 Volume Rendering

Volume rendering is a generic term for a set of techniques for visualising volumetric data and includes methods such as volume raycasting [16], shear warping [17] and splatting [18]. Volume raycasting is of particular interest to this project as recent developments in graphics card technology (enhanced programmability and new architectures suitable for general-purpose computation) can leverage this technique. Furthermore, it has already demonstrated its effectiveness during medical case studies done as part of the MARS project [19, 10].

In volume raycasting, the dataset is treated as a gaseous cloud and the representation of this cloud is simulated based on a physically plausible lighting model, as seen from the user's position. A camera is placed at the user's position and one ray is cast through the camera screen (the *image plane*) for every pixel of the image being rendered (Fig. 2.6). Rays travel through the volumetric dataset and accumulate colour and opacity by sampling raw data at regular intervals (Fig. 2.7). The interval between samples determines the quality of visualisation and affects the execution speed of the visualisation application. A smaller interval (more samples per ray) results in better visual quality at the expense of performance, while a larger interval (fewer samples per ray) may lead to perceptible artifacts but will increase the speed of visualisation.

Sampling within a discrete dataset requires interpolation: for example, nearest neighbour, trilinear [20], or using cubic B-splines [16]. Higher quality interpolation schemes take into account a larger portion of the dataset (that is, interpolate from

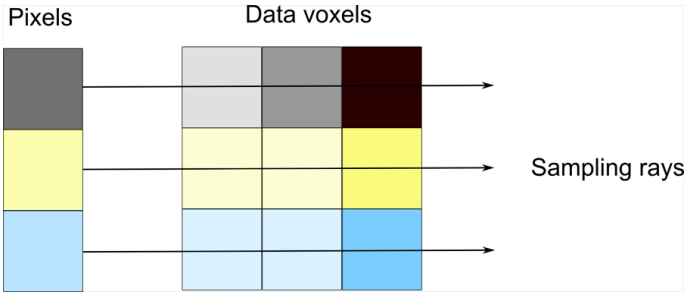

Figure 2.7: Sampling and colour accumulation during volume raycasting - raw data is sampled at regular intervals along each ray and the value at each sampling point is added to the sum over the entire ray. When a ray terminates, the colour it accumulated is displayed as an image pixel (column shown on the left).

a larger number of neighbouring voxels according to certain metrics or weights) when calculating the sample value at a point. This results in slower performance, but generally also provides smoother surfaces and fewer visible image artifacts.

The choice of the interpolation algorithm affects both the visual quality and performance of visualisation software [21]. For time-critical applications, trilinear interpolation has been found to be a good choice due to being the least resource-intensive interpolation technique [22]. A detailed description of this method is given in section 4.3.2.

Colour and opacity can be assigned to raw sample data through the use of a transfer function [23]. Essentially, it is a mapping between grayscale sample values and RGBA colour values. A well-designed transfer function can be used to remove empty space while simultaneously emphasising regions of interest. Fig. 2.8 illustrates the use of a transfer function to visually separate different regions of the volumetric dataset.

Once a sample is taken and classified by the transfer function, it is added to the sum over the entire ray in a process called compositing. Essentially, the contribution of each individual sample to the total colour and opacity is decided. This contribution is dependent on opacity of volume data, as set by the user, and the maximum number of samples. For example, if the maximum number of samples is high, then a single sample will be multiplied by a smaller constant than if the number of samples was low. A sampling ray terminates if it leaves the boundaries of the volumetric dataset, accumulates enough opacity (for example, 99% of the maximum possible opacity) or reaches the maximum number of samples.

Acceleration techniques can be used to minimise sampling in empty regions (that is, those regions classified as empty by a transfer function or by comparison

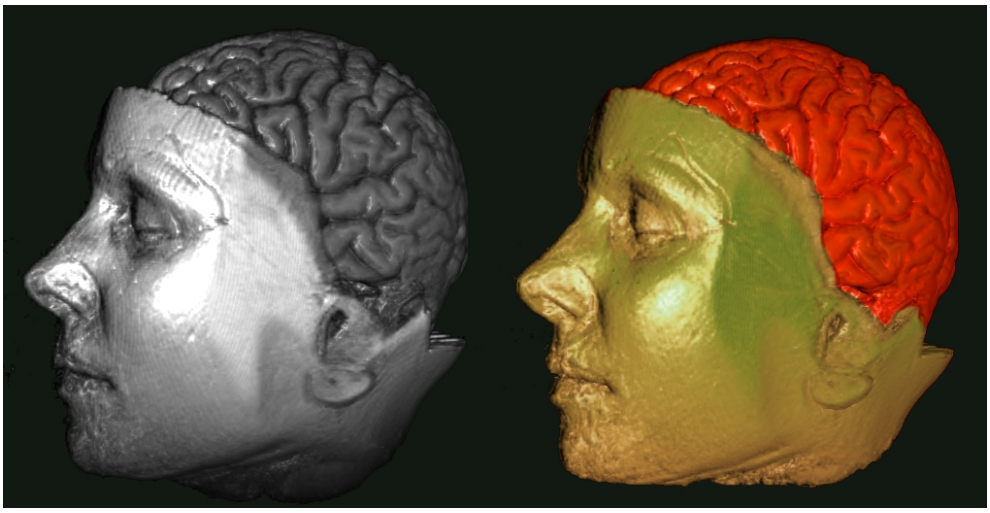

Figure 2.8: The MRBrain dataset rendered with (right) and without (left) a transfer function that assigns colour and opacity to sample values. A good transfer function can help users analyse data by visually segmenting it.

with a threshold value) [24]. The implementation of acceleration techniques in a spectral CT data visualisation application is considered in section 4.4.

2.2.3 Visualisation of MARS-CT Datasets

Spectral CT scanners (such as the MARS-CT system) are capable of generating several large datasets per scan and therefore 3D visualisation is even more complex than that of standard CT data. Generally speaking, there is no upper limit on the size of a spectral CT dataset or the number of energy bins in it, as the scanner can be programmed to arbitrarily partition the x-ray spectrum. This presents serious challenges during processing and visualisation.

Spectral CT data is not equivalent to 3D textures or other 3D volumetric datasets, but it shares some similarities with these data types. Hyperspectral data obtained, for example, during geological scanning using ultrasound [25], 4D echocardiograms [26] and PET-CT scans, where CT and PET datasets are acquired simultaneously to avoid dataset registration problems are perhaps most similar to spectral CT data. For a more in-depth look at the similarity of spectral CT datasets to other data types from the point of view of visualisation, the reader can be referred to the Masters thesis by de Ruiter [19].

Spectral CT datasets, such as the ones acquired by the MARS-CT system, are similar to standard CT datasets and can be visualised as such, if only one energy bin is viewed at a time. However, this method does not take advantage of the

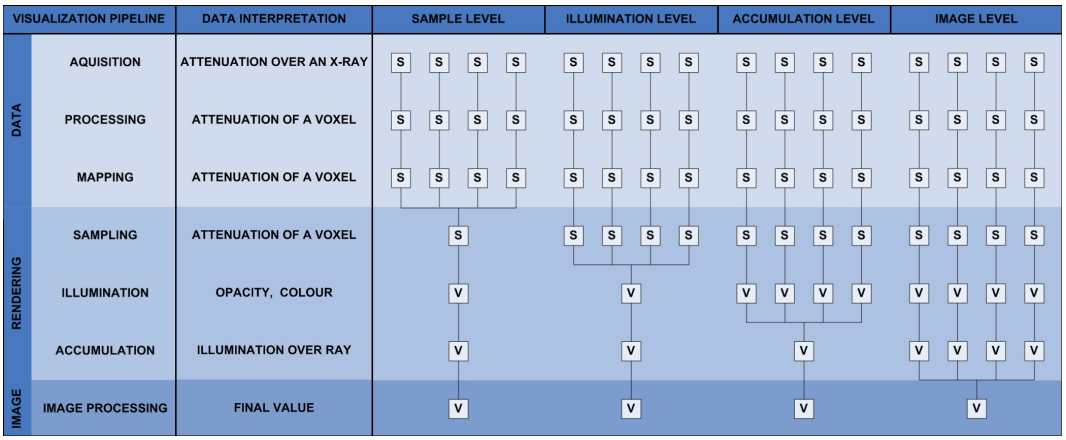

Figure 2.9: The principle of intermixing as it applies to spectral CT data. Data from different energy bins can be combined during the sampling, illumination or accumulation stages. Image courtesy of Niels de Ruiter.

unique properties of spectral CT technology, namely its ability to discriminate between body tissues or contrast agents. In order to fully explore the possibilities offered by spectral CT, energy bin data must be *combined* in some fashion to bring out the details present in the full multi-energy bin dataset.

Algorithmic combination of spectral CT energy bins is an important open-ended research topic that is currently being investigated by the MARS-CT visualisation team. The main benefit of spectral CT is its ability to distinguish between contrast agents or tissues in the body. However, spectral CT datasets are a novel data type (see section 3.1 for further discussion) and there are no established methods for processing and visualising them in such a way as to leverage their specific properties.

One approach is to use the real-time intermixing concept: the user is given a set of commands which may be used to combine different datasets at different stages of the visualisation pipeline [27]. If the framework for intermixing is flexible and generic enough, the user can experiment with a variety of combinations, rapidly identifying and discarding inappropriate ones and further investigating those that show promise. This principle is illustrated in Fig. 2.9.

The thesis by de Ruiter [19] focuses on this aspect of visualisation of spectral CT datasets produced by the MARS-CT system. His application, MARSCTExplorer, is a visualisation framework for spectral CT data that is based on OpenSceneGraph, OpenGL and utilises the GLSL shader language. MARSCTExplorer allows the user to mix energy bins using 15 different operators (such as addition, subtraction or multiplication) in real-time. The resulting combinations, if done

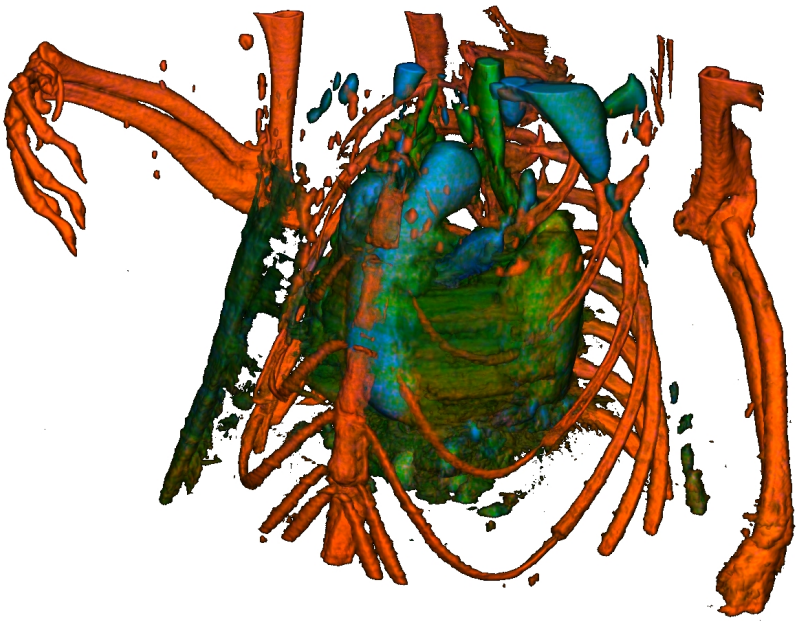

Figure 2.10: Mouse12 dataset visualised using MARSCTExplorer. Materials are visually separated through the use of colour: calcium shown as orange, barium as blue and iodine as green. Image courtesy of Niels de Ruiter.

properly, can present the data to the user in a much clearer fashion and have already been used to distinguish between different contrast agents injected into mice [10]. Fig. 2.10 illustrates how intermixing of spectral CT data may be used do distinguish between different materials. For further information, see Chapter 5 of de Ruiter's thesis.

The downside of real-time intermixing is that all energy bins must be kept in a fast storage medium because any voxel in the full spectral CT dataset may need to be retrieved at any time and may be used a number of times as part of a complex sequence of combinations. The algorithms created over the course of this research aim to be compatible with a real-time intermixing framework such as the one created by de Ruiter.

In addition to intermixing, spectral CT datasets can be processed with real-time PCA [28] (*principal components analysis*, a statistical technique for analysing multi-dimensional data) to attempt to find and separate different materials. Results indicate that three components identified by PCA for a dataset of a mouse can be used to correctly distinguish between calcium, iodine and barium [29].

Figure 2.11: Difference between architectures of a typical CPU and GPU. Diagram from the NVIDIA CUDA C Programming Guide Version 4.0 [35].

2.3 GPGPU Technology and CUDA

GPGPU (General Purpose Computing on Graphics Processing Units) is a technology that enables a large amount of scientific and industrial computation to take place on inexpensive consumer-grade GPU hardware [30]. Essentially, it involves utilising a mass-produced GPU as a co-processor for certain parallel tasks. Such tasks are, for example, matrix multiplication, n-body simulations [31], fast Fourier transforms (FFT's), volume rendering [32, 33] and video encoding [34]. In general, any task with a large number of components that can be run in parallel, independently of one another, is a good candidate for GPGPU acceleration.

GPUs have specialised architectures that are highly adapted for massively parallel processing of independent blocks of data by performing the same logical or arithmetic operations on different locations in memory. In order to quickly render complex scenes such as those found in modern PC games, per-vertex and per-pixel operations (generally on 32-bit floating-point data) must be performed billions of times per frame and thus GPU performance is commonly measured in floating-point operations per second (FLOPS).

As a consequence of having specialised architectures, GPUs are able to perform extremely well in a particular class of tasks that include a large amount of parallel computation. However, they may not suitable for general purpose computing tasks that involve a signle thread performing sequential computation. If compared to CPUs on a hardware level, GPUs have a proportionally higher number of arithmetic logic units (ALUs) and floating-point units (FPUs) but contain fewer sophisticated branch prediction units and have caches of smaller size (Fig. 2.11). Fig. 2.12 illustrates the typical architecture of a modern GPU.

This is both the key feature and the principal limitation of GPU computing.

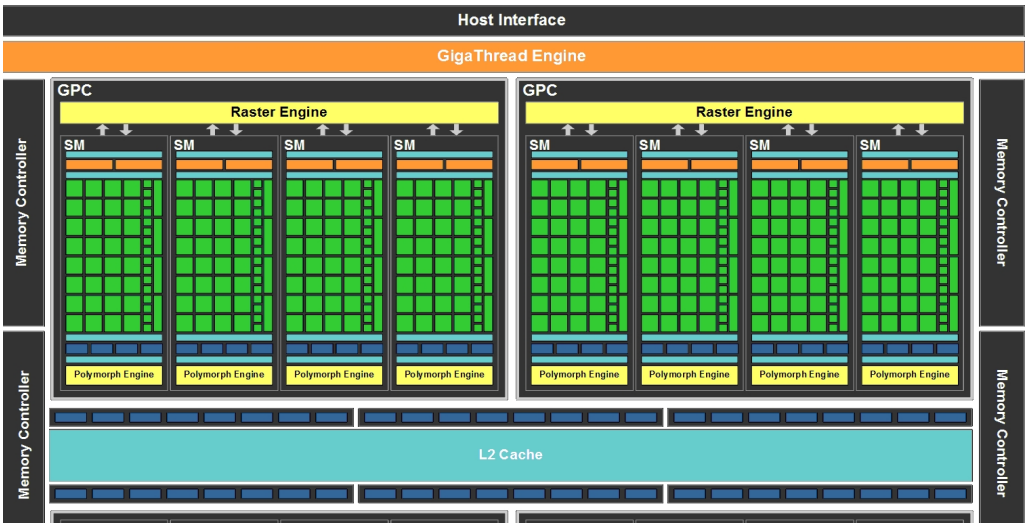

Figure 2.12: Architecture of the Fermi series of graphics cards by NVIDIA [36, 37]. Multiple cores grouped into streaming multiprocessors (SMs) are shown, along with two levels of caching and specialised auxiliary units.

GPUs are designed for parallel execution of a large number of identical commands on different data and exceed CPUs in terms of peak floating point operations per second (FLOPS). However, this speed is rarely achieved by most workloads, as supplying a GPU's numerous small execution cores with data is frequently a limiting factor, along with possibilities of branching and divergence, which, as mentioned above, are not handled well.

Until recently, there was a limited number of ways of harnessing the computational power of GPUs. Graphics languages OpenGL and DirectX have been used [24, 38, 39, 40] to process and visualise scientific data with some success, but the limited programmability of GPUs (through the use of shaders) and their unusual architectures have restricted their utility and have narrowed the potential range of problems that can be solved with the aid of their significant computing power.

While GPGPU was utilised on a small scale from the early 2000's, the real breakthrough came in 2007, with the release of NVIDIA's Compute Unified Device Architecture (CUDA) GPGPU programming language. Since that time, several GPGPU languages have been introduced with varying degrees of success, but the overall trend has been to improve GPU architectures to make them more suitable for general-purpose computation. Various methods have been used to achieve this, from addition of larger caches to improvements in programmability and ease of use. This research takes advantage of the latest features of the CUDA programming language and of the enhanced GPGPU capabilities of NVIDIA's Fermi line of

graphics hardware [37].

2.3.1 Overview of CUDA

To illustrate general GPGPU programming principles and limitations of GPUs, an overview of the CUDA programming model will be given first. CUDA is a standard GPGPU programming language, allowing the user to utilise compatible NVIDIA GPUs for general-purpose computing. While some limitations exist, most code, sequential or parallel, can in theory be re-written in CUDA. However, a substantial increase in performance against CPU code will only be observed for parallelisable tasks and very well-optimised algorithms.

The main principle of the CUDA language is that so-called *cores* (separate execution units) within a GPU are made to execute user-defined functions, called *kernels*, in parallel. A thread is one instance of the kernel function that performs operations on data located in the memory of a GPU. A large number of threads is executed at once, normally operating on different locations in GPU memory. This is referred to as the SIMT (Single Instruction, Multiple Thread) model, which is a variant of the well-known SIMD (Single Instruction, Multiple Data) model. CUDA also supports conditional branching for threads, although this normally leads to poor performance, as described later.

CUDA-capable hardware is classified by its Compute Capability (CC) that determines which functions a device supports and which architecture it uses. There is substantial variation between devices of different Compute Capabilities that spans from the number of registers a thread can use to cache size and floating point standard support. This research deals primarily with programming CC 2.0 devices and optimising code for this specific architecture. The difference between older and newer devices is rather significant, as will be demonstrated in section 5.4.

2.3.2 Kernel Launching, Occupancy and Block Size

In CUDA, kernels are launched by designating a *grid* of *blocks* (see Fig. 2.13), with each block consisting of a certain number of threads. Threads within blocks can communicate to each other and synchronise their execution using barrier commands [41]. Normally, a GPU will not have enough resources to launch all blocks in a grid at once, and thus block scheduling is needed. The internal scheduler present in all CUDA devices determines which blocks should be executed and which should be delayed and assigns each active block to a streaming multiprocessor (SM) along with a number of other blocks, depending on kernel complexity

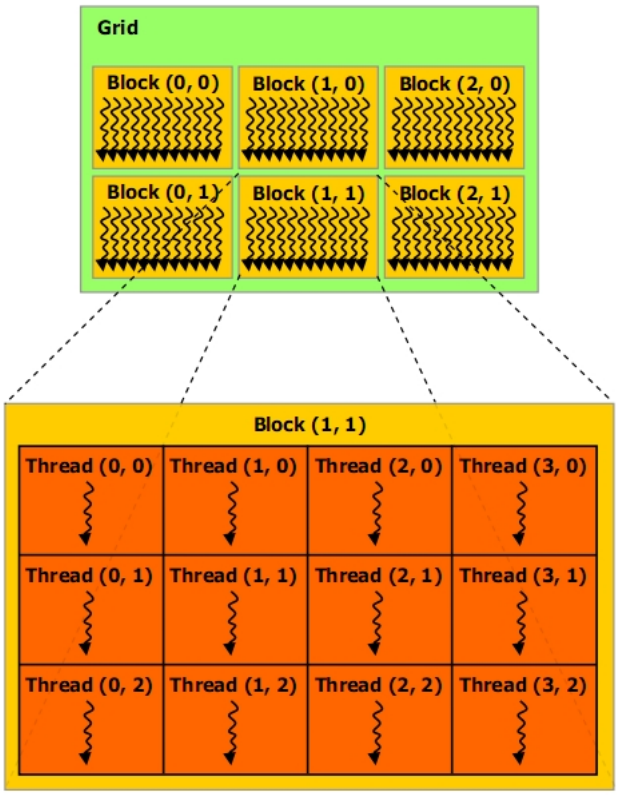

Figure 2.13: A CUDA kernel is launched as a grid of blocks, each containing a certain number of threads. Diagram from the NVIDIA CUDA C Programming Guide Version 4.0 [35]

and resource utilisation.

The number of blocks residing on an SM at one time is governed by the amount of shared and register memory utilised by each block, as these two memory types are shared between all blocks on a multiprocessor. The total number of threads used by all blocks on an SM can be expressed as a percentage of the maximum, called *occupancy*. For example, on a multiprocessor with a maximum capacity of 768 simultaneous threads, five active 128-thread blocks would constitute 83% occupancy [42].

Optimising block size to increase occupancy is important as it leads to appreciable performance benefits [32]. This is because a higher number of blocks occupying an SM allows the scheduler to hide memory access latency by executing one block while another is waiting for data from global memory [35].

It is important to keep in mind that blocks of threads on a multiprocessor may be at different stages of execution, with some nearing completion and some only

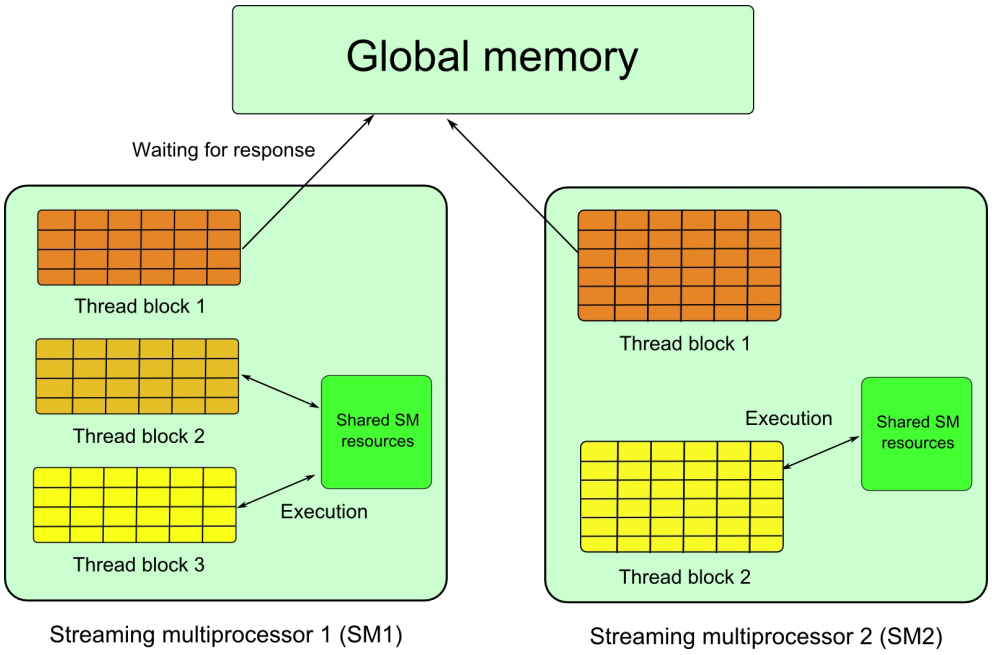

Figure 2.14: Occupancy and latency hiding mechanisms in CUDA.

beginning their work. Therefore, their requirements will be different: one block may be paused, waiting for data from global memory (an access that takes 200-700 clock cycles, depending on the architecture), while another may be active and performing fast arithmetic operations on register memory. This is the mechanism that allows latency hiding to work.

For example, in Fig. 2.14, on SM1, block 1 is waiting for data from global memory, while blocks 2 and 3 continue to execute using data from local storage. If one of them stalls like block 1, there will still be another block that can use the available ALUs and no clock cycles will be wasted. On SM2, however, the block size is larger, meaning that only two blocks can be scheduled to execute on it at any one time. This means that if block 2 also stalls, the entire multiprocessor will be forced into an idle state until one of the blocks receives the data it was waiting for.

2.3.3 CUDA Memory Hierarchy

There are three separate locations for data storage, which, in order from fastest to slowest, are: registers, shared memory and global memory. Other locations, such as texture memory and local memory are not physically distinct from the three types mentioned above, and merely describe the difference in usage. Proper memory utilisation is the key to extracting high performance from a CUDA program, as

most kernels are bandwidth-bound and not compute-bound.

The terms compute-bound and memory- or bandwidth-bound refer to the execution speed of CUDA kernels being limited by either computational complexity (cores being supplied with enough data, but being restricted by the speed of the ALUs and FPUs) or by memory bandwidth, where the data is processed by the CUDA cores faster than it can be supplied to them over the GPU's internal memory bus. For optimal performance, a high ratio between ALU calls and memory access instructions (arithmetic intensity) needs to be maintained [43].

Registers are the fastest type of memory, located inside the die and used for storage of most data by CUDA threads. There is, however, a limited number of registers, with the maximum of 63 32-bit registers being available on modern Fermi (Compute Capability version 2.0 and 2.1) hardware. Data that is too large to store inside register memory can be placed into shared memory, which is allocated from a pool of limited size that is shared between blocks. It is, on average, 6-7 times slower than register memory [44], but the difference will vary with architecture (newer GPUs will be better optimised for operations on shared memory [45]). Fig. 2.15 illustrates the different types of memory utilised by CUDA.

Various caches are also employed at different levels, but there exist significant differences between their implementation and size on different CUDA-capable hardware. Modern CUDA architectures such as Fermi (Compute Capability versions 2.0 and 2.1) contain L1, L2 and constant and texture memory caches [35].

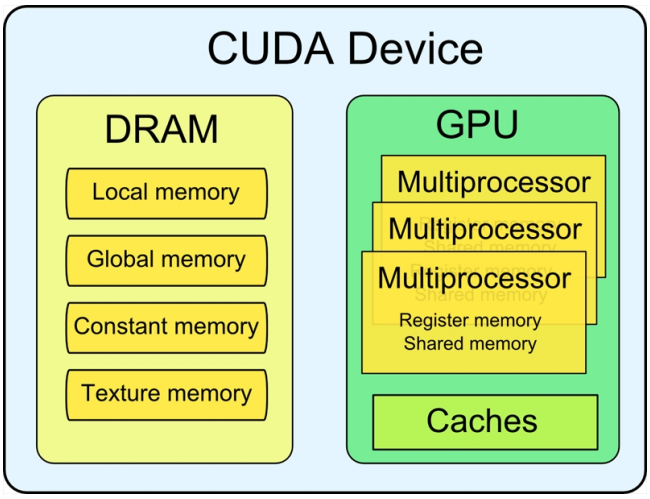

Figure 2.15: Memory hierarchy in CUDA. Access to the four memory types located in DRAM is slow, although caching of their contents inside GPU caches is possible. It is preferable to keep as much data as possible inside register and shared memory and only access DRAM when necessary.

Global memory, which utilises off-die memory chips, is by far the largest and slowest form of storage on a graphics card. Access to global memory is comparatively slow, which may lead to an imbalance where the fast ALUs and FPUs do not have enough data to operate on. As mentioned above, CUDA is specifically designed to compensate for this memory access latency. As some threads are waiting for data to be fetched from global memory, the CUDA driver schedules execution of threads from another thread block.

Normally, global memory is used for texture storage for graphics applications such as video games and therefore caches optimised for two-dimensional spatial access and linear interpolation hardware (inside specialised *texture units*) are present. These caches are automatically used and can benefit both traditional graphics applications and CUDA kernels.

Due to the limitations described above, CUDA programmers generally utilise an approach that attempts to negate or at least minimise the delays resulting from global memory access latency. The standard order of operations for most CUDA kernels is to load a small amount of data from a global memory array into shared or register memory and operate on it as much as possible, only loading other data from global memory when absolutely necessary. Large synchronised transactions initiated by all threads (for example, loading of data at the beginning of a kernel's execution) are vastly preferable to small randomly distributed fetches by a small subset of threads.

2.3.4 Conditional Branching and Divergence

In the CUDA programming model, all execution is based on warps - groups of 32 threads that are processed together by the hardware. Warps must be executed in lock-step, which requires the CUDA driver to handle divergence (from *if* or *switch* statements, for example) the following way: all threads taking one path at a divergence point are executed, while the rest are put into a waiting state. Then a different group of threads that is taking another path is scheduled for execution, and so on. This is referred to as *warp serialisation*.

In the worst-case scenario, when all 32 threads in a warp diverge, sequential execution occurs. This is in fact much worse than the same kind of divergence occurring inside code executed on the CPU due to the GPU's comparatively poor branch prediction hardware, smaller caches and lack of advanced pipelining. Fig. 2.16 illustrates the concept of warp divergence.

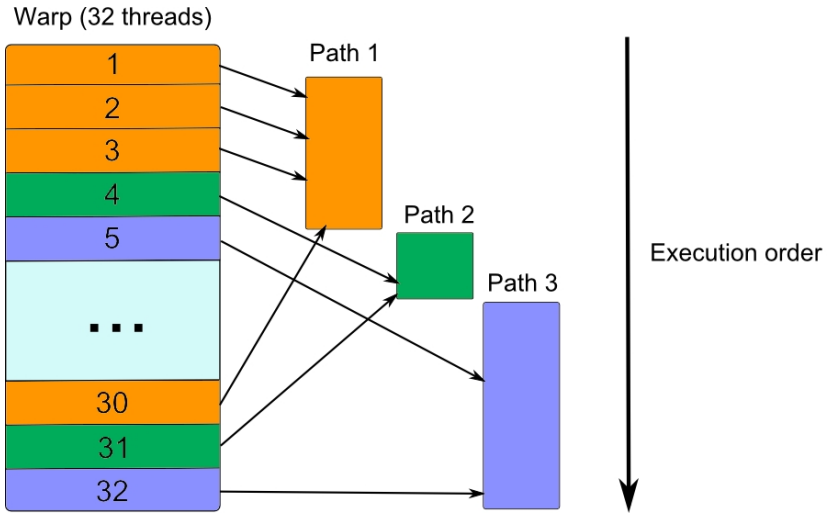

Figure 2.16: Divergence in a CUDA warp. If groups of threads within a warp take different paths, they are serialised and executed sequentially. This is one of the major causes of poor performance in CUDA kernels.

2.3.5 Volume Rendering with CUDA

As mentioned before, CUDA provides a larger degree of flexibility compared to programmable GPU shaders because it gives the programmer the ability to freely access any location in global memory from GPU code. Kernels can read from any valid location and store results in any properly declared and allocated array. In practice, this can be used to enhance the functionality of any complex algorithm. This section explains how novel features introduced by CUDA have affected the field of GPU-accelerated volume rendering.

The first instance of CUDA being used for volume rendering occurred when NVIDIA released version 2.0 of the CUDA SDK, which included a sample implementation of a volume raycaster. Marsalek et al. have improved upon NVIDIA's existing design by optimising threading and memory usage, which resulted in a 68-114% performance increase [32]. Their work demonstrates the importance of in-depth knowledge of the chosen hardware platform and the fact that even minor optimisations can dramatically alter the execution speed of an algorithm.

Kainz et al. demonstrate some unique advantages of using CUDA for volume rendering [33] by modifying empty space skipping, an octree-based acceleration technique, to make use of CUDA's ability to read from and write to arbitrary buffers in GPU memory. In their paper, Kainz et al. have shown how, when a transfer function changes, an octree could be updated in real-time on the GPU with minimal intervention from the CPU. Such an operation would not be possible

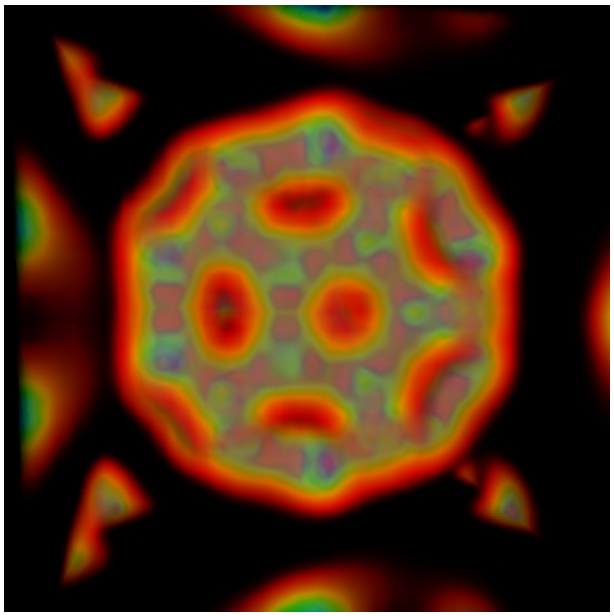

Figure 2.17: The Bucky Ball dataset (32^3 voxels) rendered by the volume raycaster that is included in the CUDA 4.0 SDK [46].

with older methods based on using shader languages to perform volume raycasting.

Smelyanskiy et al. [47] study the visualisation of medical data using high quality volume rendering algorithms on CPU and GPU architectures and offer improved implementations on both platforms. In particular, their CUDA implementation of a volume raycaster was 5-8x faster than an unoptimised version. It was, however, 12-25% slower than a heavily optimised fragment shader-based volume raycaster that took advantage of hardware rasterisation to perform empty space skipping [24].

So far, CUDA had not been used to visualise spectral CT data and this thesis is the first work that explores this possibility. Chapter 4 will discuss the design

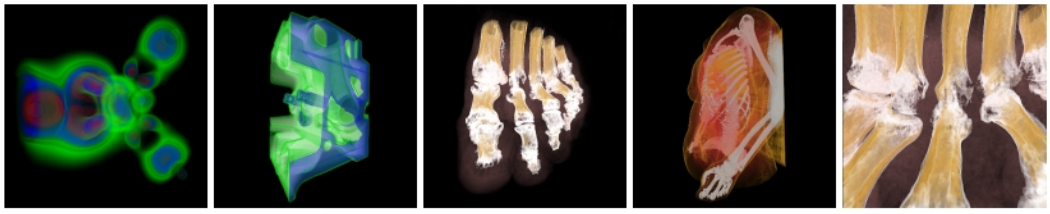

Figure 2.18: Images generated by a CUDA volume raycaster running at interactive frame rates (Marsalek et al., 2008 [32]).

of a CUDA-accelerated volume raycasting application for visualising large compressed spectral CT datasets along with certain acceleration and image processing techniques.

2.3.6 Summary

This overview, while written specifically for NVIDIA's CUDA programming language, is in fact applicable to a wide range of emerging GPGPU technologies. OpenCL [48], for example, also utilises the same general threading model but is not restricted to NVIDIAs graphics cards. In general, GPGPU programming will follow the same model in the foreseeable future and thus the general principles will remain unchanged.

In conclusion, CUDA is an excellent technology to apply to many parallel tasks such as volume rendering. The expected performance gains, for example, range from 8x-40x for fast Fourier transforms to 270x for solvers of the sum-product problem versus optimised CPU implementations [49, 50]. Iterative reconstruction of tomographic images using CUDA has yielded gains of 71x for forward projection and 137x for backward projection [43]. Another study, however, has reported more modest gains of 2.5x on average for a set of parallel computing tasks [51].

In general, the benefits of GPGPU will outweigh the difficulty of learning to correctly apply this technology to solve computational problems in many areas of science and industry. However, unoptimised CUDA code is extremely inefficient and, therefore, this research focuses not only on creating compression and rendering algorithms that run on GPUs, but also on their optimisation and tuning for particular platforms and architectures.

2.4 Real-time Rendering from Compressed Volumetric and Texture Formats

The goal of this project is to study visualisation of compressed spectral CT data, but, due to the novelty of this data type, no direct research has been conducted in this area. Therefore, compression and visualisation of related data types must be examined in order determine the most suitable techniques and provide a basis for the development of new algorithms.

As mentioned in section 1.1, there is a number of requirements that any compression/decompression scheme developed during this project must satisfy. These requirements sometimes limit the range of techniques that can be utilised: for example, the need for real-time decompression on a GPU excludes adaptive dictionary methods such as LZW, or algorithms with relatively high decompression costs

such as DCT (Discrete Cosine Transform) or Fourier transform-based techniques.

This requirement points to the need for computational asymmetry, that is, the disparity in complexity (and, therefore, time taken) between compression and decompression of data. It is in fact a highly desirable feature for applications requiring real-time decompression. Block truncation coding and vector quantisation are two examples of asymmetrically complex compression algorithms that are also well-suited for random block-based access, which is essential for a 3D dataset compression scheme [52]. This section reviews some of the standard and recent compression techniques and discusses their suitability in the context of spectral CT data compression.

2.4.1 Block Truncation Coding

Block truncation coding (BTC) [53] is a compression technique originally developed in the 1970's to compress 2D grayscale images. A wide variety of implementations and improvements has been proposed over the years, but, on a basic level, BTC relies on the assumption that there is a high likelihood of pixels in a small region having similar intensity values. This property can be used to compress an image, as blocks of pixels can be represented in a more compact fashion.

The BTC algorithm can be applied to any block of pixels and involves dividing an image into a number of small subsets (blocks), calculating the mean and standard deviation of the values in each block and storing each pixel as a single bit (Fig. 2.19). The value of this bit determines whether the intensity value at this point is one standard deviation above or below the mean. Decoding a pixel involves adding or subtracting the standard deviation from the mean, based on whether its bit mask value is 1 or 0. The mean and variance of a block are preserved, but individual values may undergo significant changes.

Several newer texture compression schemes can be considered an extension of BTC: Color Cell Compression [54] and DXT/S3TC [55, 56]. BTC can be adapted to 3D blocks of voxels for compressing volume data, but the only way to achieve high compression ratios is to increase the block size, which can lead to undesirable artifacts in compressed images. Images with smooth gradients, such as photographs, can be encoded with reasonable fidelity, but the quality of compression of images with sharp contrast and thin lines will deteriorate significantly.

Spectral CT datasets could theoretically be encoded with BTC: it is asymmetric, well-suited for grayscale data with smooth gradients and should achieve the necessary compression rate. However, while it is possible to use BTC to compress individual energy bins, there is no clear way to exploit inter-energy bin correlation using this technique.

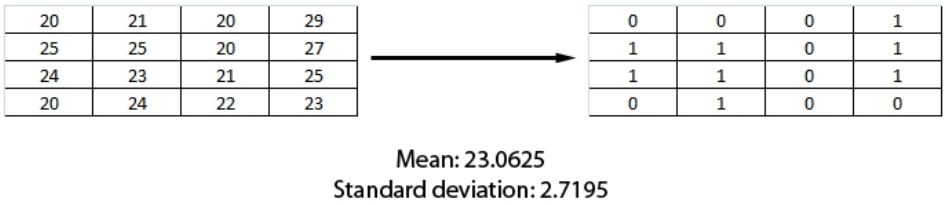

Figure 2.19: A block of 4x4 pixels encoded by BTC. Values below and equal to the mean are stored as a 0, while values above the mean are stored as 1. Therefore, the entire block of pixels can be represented by a 16-bit mask, one average value and one standard deviation value.

2.4.2 Vector Quantisation

Vector quantization (VQ) is a very promising technique for spectral CT data compression, which involves taking an n-dimensional vector and finding its closest match in a pre-generated codebook containing a certain number of vectors of length n. This operation reduces a vector of arbitrary length to a single index into a codebook, with the compression ratio being dependent on the length of the input vector.

The codebook used for VQ may be generated based on a statistical model, or, most commonly, on a training sequence [57]. The codebook should cause the least possible amount of distortion during quantisation; in other words, an input vector should be reproduced very closely by the best-matching vector in a codebook.

A very simple example of vector quantisation is shown in Fig. 2.20. An input vector is compared to every vector in the codebook. This searching method is known as the full codebook search algorithm [58]. In this example, Euclidian distance between two points in 3-dimensional space is used to measure the distortion and the codeword that causes the least amount of distortion is selected. Only the index of that codeword is subsequently stored, and, thus, compression is achieved. In Fig. 2.20, the distance between the input vector and the second codebook entry is

$$\sqrt{(1-1)^2 + (-2--1)^2 + (5-4)^2} = \sqrt{2} \tag{2.1}$$

which is the smallest distortion possible with this codebook. Therefore, the second codebook vector is selected to represent the original input vector and the corresponding index is stored.

A distinct advantage of this technique is very rapid decompression time. Lookups into a codebook can be done quickly and efficiently in CPU and GPU code, which

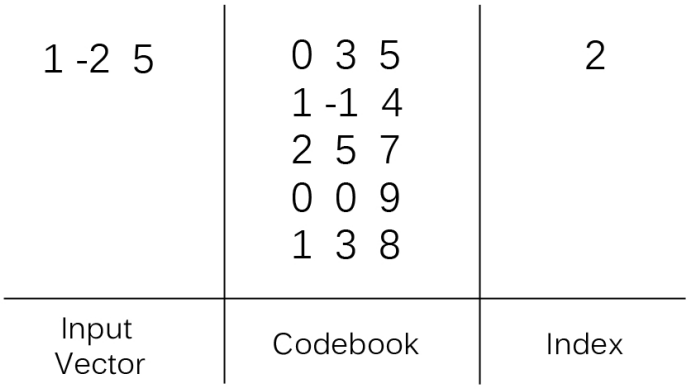

Figure 2.20: Simple example of vector quantisation.

is crucial when raw volume data is being randomly accessed millions of times per second. The advantages of VQ are not purely theoretical, however, and its benefits have long been acknowledged: a large number of papers has been published on compressing textures through the use of vector quantisation and rendering from a compressed format.

For example, a VQ-based scheme developed by Beers et al. [58] produced compression ratios of 35:1 with little quality loss. While their algorithm was only applicable to 2D RGB textures and was only adapted for execution on the CPU, the observed decrease in rendering speed attributable to compression was no larger than 20%. In addition, the authors identified vector quantisation as an excellent candidate for future texture compression algorithms due to its fast decompression speed, possibility of random access to small blocks of data, decent compression ratio and acceptable visual quality.

Subsequent studies have confirmed these observations, such as the paper by Schneider and Westermann [59], which is perhaps the work most closely related to this research. Their complex scheme based on decomposing large volumes into small voxel blocks and quantising them was successful in reducing the size of 3D RGB volumes with little performance decrease. In fact, by introducing acceleration methods such as empty space skipping, performance was increased to the point of being faster than that of the standard volume renderer used for comparison. The algorithm, however, was created for the general 3D 8-bit per channel RGB texture data type, suffered from artifacts at high compression ratios and was only restricted to nearest neighbour interpolation.

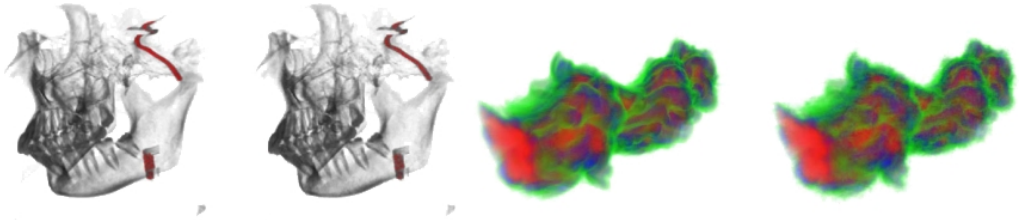

Figure 2.21: Examples of volumetric data visualised by rendering directly from a compressed format by Schneider and Westermann [59]

2.4.3 Wavelet-based Methods

The discrete wavelet transform (DWT) is a technique that involves passing a signal (or dataset) through a series of filters to obtain a multi-level decomposed representation [60]. This process, while not compressing the original data, may be used to aid subsequent compression algorithms by identifying trends and similarities in 1D, 2D or 3D space, depending on the dataset and transform used.

DWT can be used in any number of dimensions by successively applying it in each dimension. For example, Montgomery et al. [61] attempt to decompose volumetric datasets using the 3D Haar wavelet transform. The motivation is to take advantage of spatial similarity in order to eliminate redundant information in the dataset. Their algorithm uses the classic wavelet compression process - a 3D volume dataset is first decomposed by DWT and then the coefficients estimated to be unimportant (according to a threshold) are removed. This may get rid of as much as 95-99% of all coefficients, resulting in high compression rates.

The strength of wavelet transforms lies in their ability to identify spatial coherence and patterns in datasets. If an image is processed with a suitable transform and if the tresholding parameters are set correctly, then the majority of coefficients that are removed will be too small to meaningfully contribute to the image. Therefore, discarding them will help reduce the size of the image without introducing any noticeable visual distortion.

High-quality wavelet-based compression schemes have been created. Zeng et al. [26] compressed time-varying data from 4D echocardiograms with the aim of producing highly-compressed images that retain sufficient image quality, with computational efficiency not being a priority. This resulted in slow (61-210 seconds, for datasets of varying sizes) decompression speeds for the entire dataset.

Their technique provides no way of randomly accessing data, so this solution can only be considered appropriate for the purposes of long-term storage of echocardiographic datasets on permanent storage media. In fact, one such issue,

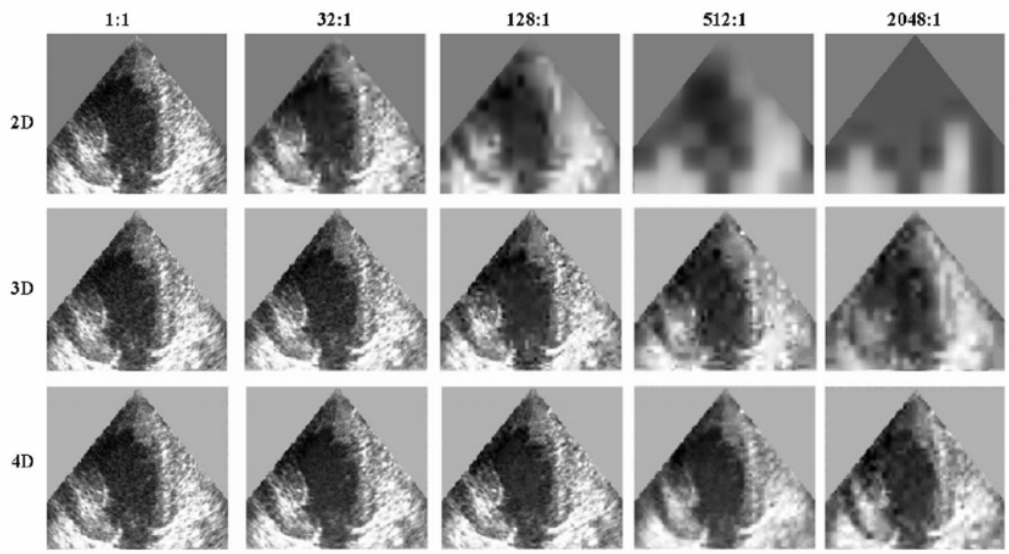

Figure 2.22: Example illustrating the advantage of compressing 4D datasets using a true 4D approach based on the 4D wavelet transform and zerotree coding [62]. Image by Zeng et al. [26].

namely the impossibility of slice-by-slice access to the dataset, is acknowledged by the authors. Their work, however, is important as it explores the use of wavelets for finding coherence in 4D datasets and exploiting it during compression. It also illustrates the limitations of wavelet transforms and the difficulty of introducing wavelet-based compression into a real-time rendering application.

Bajaj et al. [52] have also used a similar approach for compressing 3D RGB datasets: their algorithm is based on pre-processing using the 3D discrete wavelet transform followed by truncation of wavelet coefficients, quantisation and encoding using the zerobit encoding scheme which involves constructing cell bit flag tables (CBFTs) that use single bit values to specify whether a block of volume data (4^3 voxels) is empty. Such an approach significantly reduces the number of table lookups that need to be performed during decompression and reduces empty blocks to a single bit stored in memory.

2.4.4 Summary

Overall, any kind of lossy compression algorithm used in a real-time rendering application can be thought of as a compromise between three primary requirements: compression ratio, execution speed and image quality. Because of limited hardware resources, a change in one of the three factors necessarily affects the other two and thus a balance must be found. This research attempts to design and implement

algorithms that conform to these requirements and provide a platform for further research into spectral CT data visualisation.

Out of the possible compression methods, vector quantisation appears to be an excellent technique to use for decompressing large datasets in real-time, because only simple table lookups are required. It also provides a highly flexible approach to compression - the choice of vector to be quantized is entirely arbitrary, as it can be formed from any part of the dataset at the discretion of the algorithm designer.

Vector quantisation may be applied to 2D, 3D, multi-variate and time-varying data, and, in general, has very few limitations. While visual quality varies based on several factors, it is not commonly a problem at reasonable compression ratios. Due to these properties, algorithms based on this technique form the basis of both compression schemes presented in this thesis.

2.5 Image Quality Metrics and Compression of Medical Images

As mentioned previously, image compression can be lossy or lossless. Obviously, lossless compression of any medical image has no effect on its diagnostic utility, but it may be useful for reducing storage space and transmission time if the images are accessed remotely [63]. However, lossy compression is an entirely different issue that has generated a lot of debate in medical literature.

Lossy compression has the potential to negatively affect image quality, and, by extension, diagnostic accuracy. Misdiagnosis may have serious medical and legal consequences and therefore any application of lossy algorithms to medical data must be considered carefully.

While the MARS-CT scanner is still in an experimental stage and algorithms developed over the course of this research will not be integrated into software used for diagnosing human subjects, it is nevertheless useful to provide a survey of factors that influence the use of lossy compression during storage and visualisation of medical data.

In general, image quality can be assessed by objective or subjective means. This also holds true for the field of medical image compression, where objective methods such as MSE (Mean Square Error) or PSNR (Peak Signal-to-Noise Ratio) are used, along with subjective evaluation by professionals who rank images on a scale to indicate how the degradation in image quality affects their diagnostic value. The next two sections will explain the use of objective and subjective quality assessment techniques in detail.

2.5.1 Objective Metrics

Objective image quality metrics involve comparing original and compressed images or volumetric datasets and calculating a value or set of values that attempt to describe the differences between them. These metrics cannot produce entirely accurate measurements of perceived image quality, but some general information can still be gathered.

PSNR, in particular, is a commonly used method for image quality evaluation. It is based on the Mean Square Error (MSE) metric which can be applied to any two datasets to determine the difference between them. For two 3D volumetric datasets I and K, where I is the original volume and K is the volume modified in some way (for example, compressed using a lossy method), MSE is calculated according to the following formula:

$$MSE = \frac{1}{mnl} \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} \sum_{k=0}^{l-1} [I(i,j,k) - K(i,j,k)]^2 \tag{2.2}$$

Essentially, MSE is concerned with determining how different the datasets I and K are by calculating the differences between individual voxels and finding the average. The squared difference value for each voxel coordinate is added to a running total, which is later divided by the total number of voxels in a dataset. The result is a single MSE value that can be used as an image quality metric. PSNR adjusts it by taking into account the maximum possible voxel value:

$$PSNR = 20 \cdot \log_{10} \left(\frac{MAX_I}{\sqrt{MSE}} \right) \tag{2.3}$$

For certain applications, such as transmission over unreliable wireless channels, lower PSNR can be tolerated [64]. However, for precise work such as medical diagnosis PSNR must be kept as high as possible, as higher PSNR corresponds to fewer differences between the original and compressed datasets.

Quantitative methods such as PSNR calculation are reasonably simple, do not depend on the observer and can be run as many times as needed, making them ideal for eliminating some obviously unsuitable compression parameters during algorithm design. On the other hand, it may be difficult or impossible (depending on the method) to establish which values have been changed, how they have been changed and how these changes may affect the subjective human perception of image quality.

In addition to this problem, another issue may arise if the same quality metric is applied to images compressed by different algorithms. For example, the types of defects (such as visibly blocky areas or blurring) introduced by JPEG's Discrete

Cosine Transform (DCT) and subsequent quantisation may be different to those introduced by a wavelet or DXT-based scheme even though a similar objective metric may be measured if the same image was compressed with both algorithms and evaluated.

The problem of using objective metrics to evaluate image quality and the inherent unreliability of these approaches is well-explored. For example, Eskicioglu and Fisher [65] criticise MSE as being a metric that does not take into account the experience of the human visual system. Instead, the authors consider other metrics, Hosaka plots in particular, to correlate much better with subjective quality evaluations. However, Hosaka plots are not a common measurement and few papers on image compression utilise them for quality evaluation.

2.5.2 Subjective Methods

Subjective evaluation by highly trained experts is the other class of image quality assessment methods. In the field of medical image compression it was attempted by, for example, Cosman et al. [66], Fritsch et al. [67] and Savcenko et al. [68]. In these cases, subjective evaluation involved medical professionals being asked to assess image quality according to the standards considered acceptable in their specific areas of expertise.

The obvious advantage of such a procedure is that the diagnostic value of a medical visualisation technique or image compression method can be judged in this fashion. It is far more useful than a PSNR value as it takes subjective human experience into account. Such experiments, however, are reasonably difficult to organise and are time-consuming.

A study exploring the effects of image compression on the diagnostic value of medical images was conducted by Karson et al. [69]. A panel of echocardiologists examined images compressed with the JPEG algorithm (without any knowledge of such compression having taken place) and only found noticeable degradation at compression ratios of approximately 30:1, which corresponded to an SNR of 23.84 dB. However, even this relatively high compression ratio didn't render images useless for diagnostic purposes. The expert panel consulted during the study noted that even though visible artifacts were present, they did not hinder diagnosis. Other research however, such as the paper by Fritsch et al. [67] has reported medically relevant changes in JPEG images compressed at the ratio of 10:1.

Ghrare et al. [70] also utilise the same dual subjective/objective metric approach for image quality evaluation as Karson et al. This paper focused on conducting an evaluation of how objective metrics such as SNR, MSE or PSNR are related to a rating assigned to a compressed medical image by an expert in the

area. Evaluation of three medical case studies of CT images (adrenal gland, liver and chest) compressed with a discrete wavelet transform-based method was performed and numerical metrics were computed. For images deemed Excellent, Good or Fair (based on Mean Opinion Score) PSNR varied between 36.745 and 25.962 dB and the compression ratio varied between 10:1 and 30:1. These findings are consistent with those of Zeng et al. [26] that indicate that a 30:1 compression ratio may be acceptable for medical use.

Erickson [63], suggests that tolerable compression ratios from one modality (that is, a CT scan targeting a specific set of organs or tissue types such the chest or abdomen) cannot be relied upon to give an accurate impression of what compression ratios can be considered acceptable in other datasets, even ones from the same imaging modality. Erickson also notes that certain techniques used in medical imaging, such as reducing acquisition time by decreasing the number of excitations (radio frequency pulses that excite protons and cause them to switch to a higher energy state) during an MRI scan can be considered "lossy", as they result in images of lower quality without the use of compression. Overall, however, the author suggests that the widespread use of lossy compression needs to be examined further, as there appear to be no major disadvantages, provided that compression algorithms are thoroughly evaluated to ensure that the diagnostic value of images compressed by them is retained.

In conclusion, objective image quality metrics, while being limited and sometimes unreliable, are the simplest and fastest way to evaluate the quality of compression. Medical, or subjective evaluation in general, while being a much better indicator of the utility of a compression algorithm in real-world situations, is significantly more difficult to organise and conduct.

For the purposes of this thesis, medical evaluation is unnecessary, as the MARS-CT system is not being used to scan human subjects. In the future, however, a detailed evaluation of the whole visualisation toolchain would be highly beneficial as it would help validate the visualisation techniques used to present spectral CT data to the users and establish the effect of compression on image quality.

2.6 Summary

Research in several areas, such as medical imaging, texture and volume data compression and volume rendering has been reviewed in order to establish guidelines for designing compression and visualisation algorithms for spectral CT datasets. Due to the novelty of spectral CT data, techniques applied to similar data types have been considered. In conclusion, this chapter has demonstrated that:

- Spectral CT is a novel medical imaging technique that simultaneously measures the attenuation of x-rays over several energy ranges of the electromagnetic spectrum. Different materials and tissues within the body can be discriminated after spectral CT data is processed and visualised using appropriate algorithms, which may lead to improved diagnosis of conditions such as atherosclerosis or fatty liver disease.

- A real-time 3D visualisation application that renders directly from compressed spectral CT datasets can be created with currently available GPU hardware and programming languages. Previous research has achieved similar goals using more limited technology [59, 52, 71]. Modern GPU hardware possesses vastly superior capabilities and the CUDA programming language is highly advanced and well-adapted to parallel GPGPU tasks.

- Lossy block-based compression simplifies the process of visualising large datasets. Block-based methods for 3D textures and other volume data are available and can be extended to spectral CT datasets.

- Vector quantisation is able to provide random access to volume data and offers a good balance between decompression speed and image quality.

- Subjective evaluation of the quality of compressed medical images or volume data is desirable. Objective metrics such as MSE or PSNR are substantially easier to calculate, but must be interpreted with caution as they do not always correlate with perceived visual quality.

Chapter III

Compression of Spectral CT Datasets

This chapter examines the properties of spectral CT datasets and describes two asymmetrical compression algorithms for spectral CT data. As explained in section 2.2.3, during visualisation, the entire dataset must be kept in the memory of a GPU for performance reasons. The algorithms presented in this chapter are designed to compress multi-gigabyte spectral CT datasets into formats that can be quickly decompressed on a GPU.

The characteristics and unique features of spectral CT datasets are examined in section 3.1. The main requirements for a compression algorithm for spectral CT data have been mentioned previously and include the need for fast decompression speeds, ability to quickly access any voxel in any energy bin and acceptable image quality. These and other requirements are further investigated in section 3.2.

Two algorithms (VQ1 and VQ2) designed for compressing spectral CT data on the CPU and rapidly decompressing it on the GPU are presented in sections 3.3 and 3.4 respectively. This chapter describes the theory behind both algorithms, while their practical implementation in a visualisation application for large spectral CT datasets will be detailed in Chapter 4.

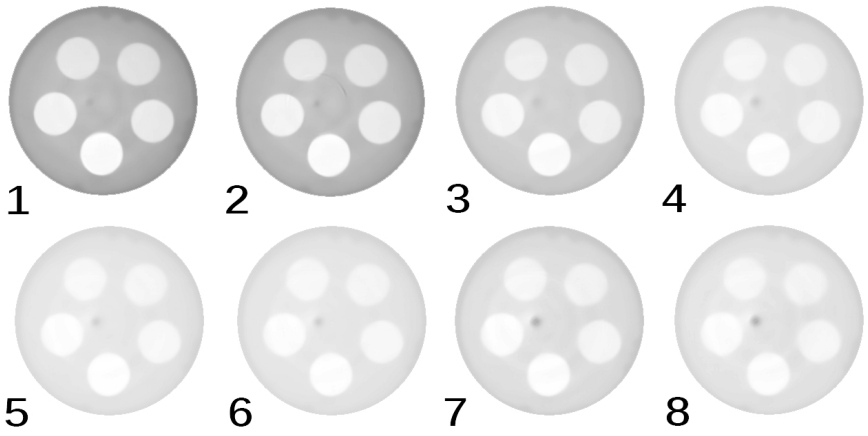

Figure 3.1: 8 energy bins of slice 10 of the Phantom dataset.

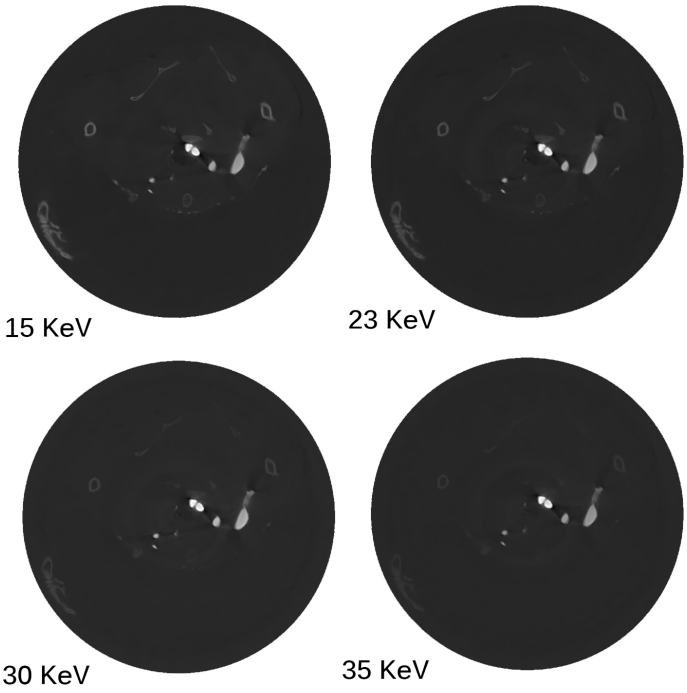

Figure 3.2: 4 energy bins of slice 423 of the Mouse12 dataset.

3.1 Properties of Spectral CT Data

As described in section 2.1, spectral CT data comes in the form of a single dataset containing several energy bins, with each bin containing the attenuation of x-rays over a certain energy spectrum as measured by a specialised detector. Each energy bin can be treated as a complete standard CT dataset, or the entire dataset can be looked at as a whole.

For the purposes of this research, energy bins are thought of simply as volumetric datasets that share a high degree of similarity, or correlation with each other. Other approaches are also possible, such as attempting to create an underlying physical model that takes into account the x-ray energies used for image acquisition. The aim of this research, however, is to create algorithms that are reasonably generic and can be applied to any form of data sufficiently similar to spectral CT datasets.

Reconstruction algorithms for spectral CT generate 8-16 bits-per-pixel (depending on acquisition mode and detector type) grayscale images, where each image corresponds to a portion (slice) of one energy bin and an energy bin consists of a number of slices, in practice generally ranging from 256 to 512. Spectral CT datasets share similarities across both 3D and 4D space, since there is correlation

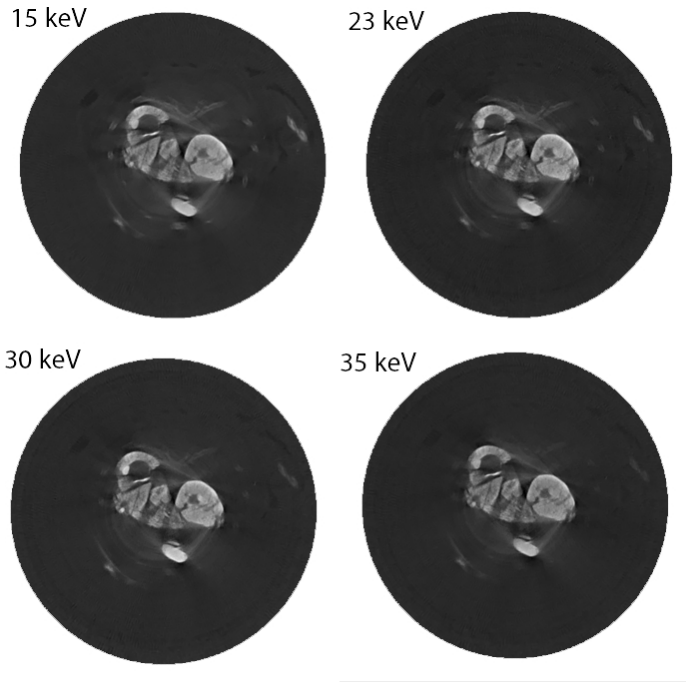

Figure 3.3: 4 energy bins of slice 192 of the Mouse12 dataset.

both between certain geometric features of a single energy bin and inter-energy bin correlation over the entire dataset. Therefore, the problem of compressing spectral CT datasets can be approached by treating them as sets of generic 3D volumes or as complete 4D datasets. This thesis explores both possibilities.

Theoretically, any compression technique for 3D volumetric data can be adapted to compress individual energy bins, as will be shown in section 3.4 where the VQ2 algorithm is explained. However, unique advantages offered by spectral CT technology only manifest themselves when datasets are looked at as a whole, and it is desirable to approach compression from the same point of view. This is the idea behind the VQ1 algorithm (section 3.3).

Voxels across energy bins are co-registered, meaning that voxel V_{xyz} in one energy bin will describe the same point in 3D space as voxel V_{xyz} in a different energy bin. Therefore, any individual scalar value in a spectral CT dataset can be accessed by four coordinates: x, y and z as spatial coordinates and i as the energy bin index.

A sample of slices from the Phantom dataset and the Mouse12 dataset are shown in Fig. 3.1, 3.2 and 3.3. The differences between energy bins are slight, yet visible. In general, there will be small variations across energy bins that, while being nearly unnoticeable during visual inspection, are nevertheless hugely

important as they form the basis of spectral CT imaging.

3.1.1 Dataset Size

A standard spectral CT dataset may contain any number of energy bins, although, due to the difficulty of extracting and visualising relevant information, it is unlikely that more than eight will be studied at any one time [19]. The current maximum dimensions of a slice are 1536x1536 pixels, with 512 slices per energy bin being the standard due to the technical limitations of the current iteration of the MARS-CT scanner. Slices are generally cropped to remove empty space and downsampled to around 1024x1024 pixels before being visualised. All data can be assumed to be in a 16 bits-per-pixel format. Therefore, a typical large spectral CT dataset will consist of:

$$1024 \text{ x } 1024 \text{ x } 512 \text{ x } 8 \text{ energy bins x } 2 \text{ bytes per voxel} = 8,589,934,592 \text{ bytes} = 8 \text{ GB} \tag{3.1}$$

Datasets below 1GB in size can be easily visualised with the current generation of graphics hardware, but, as the calculation above shows, a realistic spectral CT dataset can have a size of 8GB, making it too large to be visualised at its native resolution. In the future, spectral CT datasets will increase in size as the MARS-CT scanner is developed further.

3.1.2 Correlation Across Energy Bins

One important property of spectral CT datasets is inter-energy bin correlation. As seen, for example, in Fig. 3.3, there is a high degree of similarity between all four energy bins of a single slice of the Mouse12 dataset. Nearly all features are present in all four images, with only minor variations in pixel intensities.

Histograms of the energy bins of this slice may be created, showing the mean and standard deviation of pixel values. These can then be compared with the histograms of *difference images*, which are formed by choosing the first energy bin as a base and subtracting the other three energy bins from it. Several histograms of representative spectral CT slices are shown here. In all cases, the largest possible differences are shown (that is, the energy bin measuring the least amount of energy is subtracted from the energy bin measuring the most amount of energy).

Fig. 3.4 shows the histograms of four energy bins of slice 192 of the Mouse12 dataset. Note the mean and standard deviation and compare them with those obtained for the difference images of the same energy bins, as shown in Fig. 3.5. The

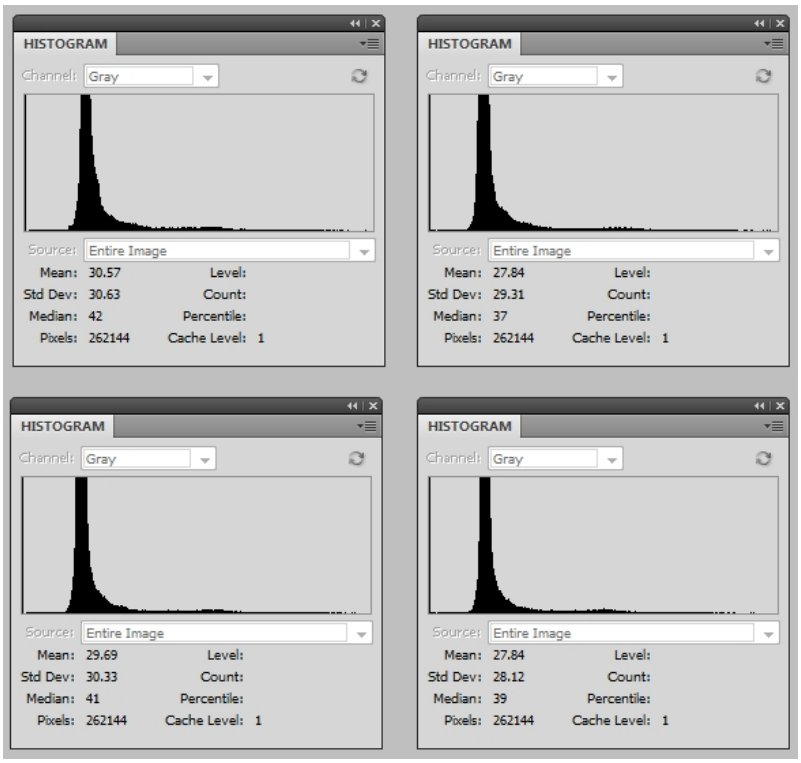

Figure 3.4: Histograms of the four energy bins of slice 192 of the Mouse12 dataset. Clockwise starting from top left: 15 keV, 23 keV, 30 keV and 35keV energy bins.

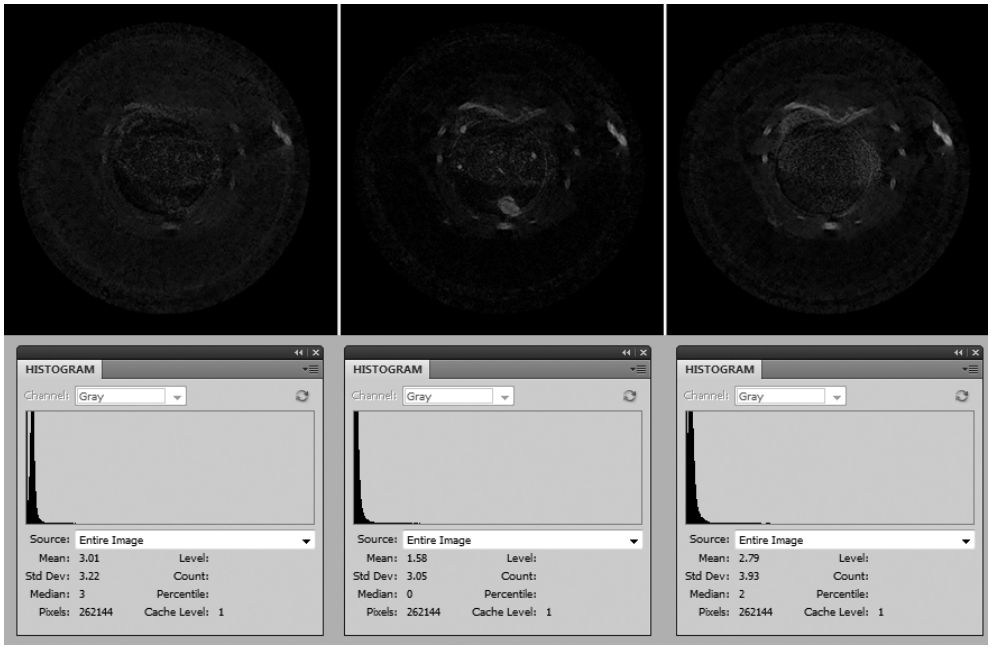

Figure 3.5: Difference images of slice 192 of the Mouse12 dataset and their respective histograms. From left to right, difference images between 15 keV and 23 keV, 15 and 30 keV and 15 and 35 keV energy bins.

visible differences are so minor that contrast and brightness had to be enhanced in these images (original versions can be found in Appendix 1). This image further illustrates the need to use custom algorithms to process spectral CT data, as differences between energy bins can be easily missed during visual inspection. There is, however, a lot of redundant information present in spectral CT datasets, and a compression algorithm working on the differences between energy bins can be envisaged.

The Plaque 1 dataset can be used as another example of inter-energy bin correlation. As seen in the histograms in Fig. 3.6, the variation among the pixels of the difference image is significantly lower when compared to the original slice. The standard deviation has been reduced from 66.72 to 20.77 and only 2.5% of pixels lie outside of three standard deviations from the mean. These high values are likely to be caused by noise, as normally the differences between energy bins are consistent and change smoothly.

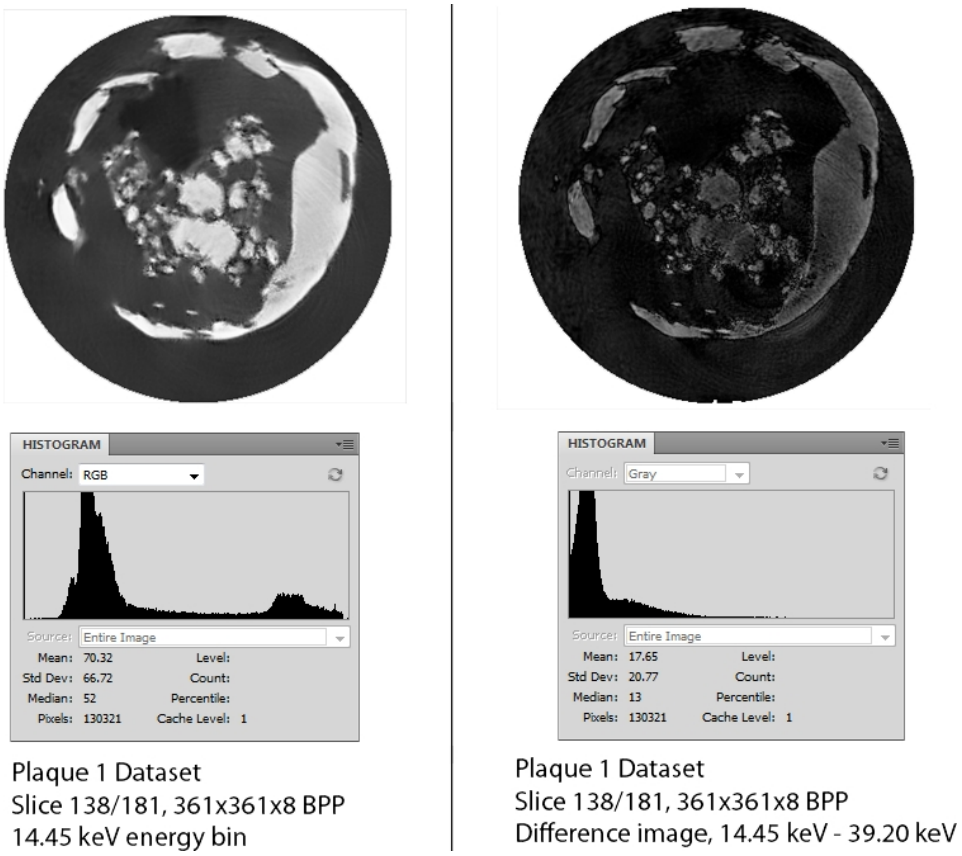

Figure 3.6: 14.45 keV energy bin of slice 138 of the Plaque 1 dataset, the 14.45 keV - 39.20 keV difference image and their corresponding histograms.

From the point of view of visualisation, a typical spectral CT dataset, just like standard CT and other medical datasets, contains some data that is not being studied and only obscures the real areas of interest to a scientist or physician. During rendering, these regions can be treated as empty space and skipped entirely, effectively reducing the number of samples that needs to be taken. Appropriate data structures and algorithms can be utilised to partition the volume dataset and reduce the sampling rate inside empty regions. These techniques will be discussed in section 4.4.

3.2 Requirements

A compression algorithm for spectral CT data needs to possess certain attributes in order to support fast, real-time decompression for the purposes of volume rendering. These guidelines are summarised below and both algorithms presented in this chapter are designed to adhere to them. The most important points relate to *decompression* rather than compression, because, during visualisation, this part of the algorithm will need to be executed when rendering each frame. Compression, on the other hand, only needs to be done once.

1. Compression can take any form, as long as it is able process multi-gigabyte datasets on standard desktop PC hardware in a reasonable amount of time. Otherwise, there are no specific requirements that a compression algorithm needs to satisfy as it will be executed on the CPU prior to visualisation.

2. Decompression must allow for fast random access to individual voxels within a spectral CT dataset.

3. Decompression algorithms must be well-suited for execution on the massively parallel architecture of GPUs.

4. Decompression algorithms must be easy to integrate into a volume rendering application for spectral CT datasets and must be fast enough to allow for interactive frame rates.

5. Sufficient visual quality for scientific study needs to be preserved.

Only a small subset of available techniques satisfies these requirements. Of these, vector quantisation (section 2.4.2) is the most flexible and promising one.

3.3 The VQ1 Algorithm

This section describes VQ1, the first of the two compression algorithms developed over the course of this project. It relies on a well-known technique of vector quantization, applied to processed spectral CT datasets and utilises the inter-energy bin correlation described in section 3.1.2.

3.3.1 VQ1: Pre-processing

First of all, a spectral CT dataset consisting of k energy bins is broken up by designating one energy bin (generally the broad spectrum bin which contains the attenuation over the entire measured spectrum, but any bin may be used) to be the base volume, and the other $k-1$ bins as the volumes to be processed by a difference algorithm. A set of difference volumes is then formed by subtracting every voxel of every energy bin from the base bin:

Algorithm 1 Calculate difference volume for a dataset of k energy bins.
Note: indexing begins at 0, where Bin^0 is the base energy bin.

for $i = 1 \rightarrow i = k$ **do**
  for all voxels j in Bin^i **do**
    $Diff_j^i = Bin_j^0 - Bin_j^i$
  end for
end for

When difference volumes are merged into a single one, the result is the formation of a new volumetric dataset with vectors of length $k-1$ at every voxel coordinate. Obviously, this process, taken on its own, does not reduce the size or the quality of the dataset in any way, as the original data is fully recoverable using an inverse operation. However, the difference volume generated with this algorithm provides a basis for the next operation. This process is illustrated on the left side of Fig. 3.7.

3.3.2 VQ1: Vector Quantisation and Codebook Generation

Two tasks need to be completed in order for the pre-processed difference volume to be compressed. A codebook needs to be generated which is then used to quantise the difference volume. Codebooks are reusable and only need to be generated once for a given spectral CT dataset.

The codebook is generated with the Linde-Buzo-Gray (LBG) [57] algorithm. This algorithm requires a training sequence larger than the desired number of

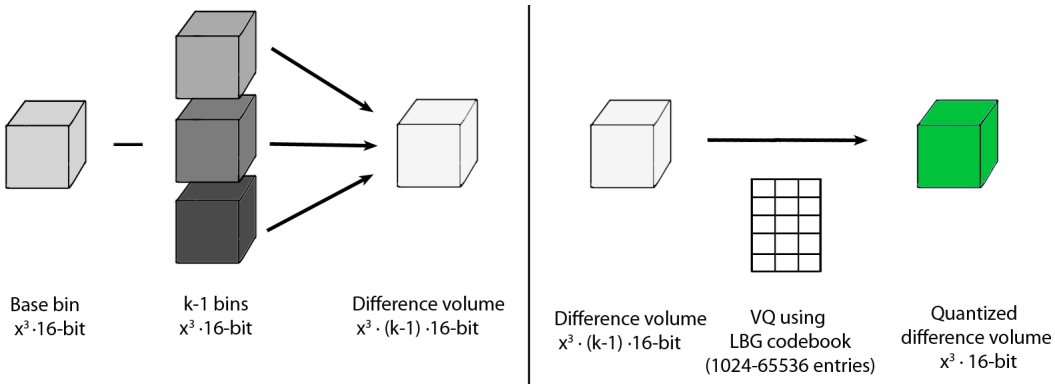

Figure 3.7: The VQ1 algorithm, as applied to a spectral CT dataset of k energy bins. A difference volume is formed by designating one bin as a base and subtracting all other energy bins from it. It is then compressed by vector quantization.

vectors in the codebook. To ensure a representative sample, 10% of the dataset is randomly chosen to serve as a training sequence (although this percentage is adjustable by the user). For example, for an 8-bin 512^3 dataset, this would mean using $13,421,772$ vectors of length 7. An average codebook might comprise, for example, 1024, 2048 or 4096 vectors.

Codebook generation is a sequential process that first attempts to find the two vectors that best describe a given training sequence, then expands to four vectors and so on, until the number of vectors (which has to be a power of two) specified by the user is reached. Each iteration completes when a certain minimum acceptable distortion threshold is reached. In other words, if the total distortion (that is, the total error introduced by approximating the entire training sequence with vectors from the codebook) is less than a certain pre-defined number, then the codebook is considered acceptable. If not, another pass over all the values of the training sequence is made that attempts to improve the fit, and so on.

The time to generate codebooks at various quality settings is examined in section 5.1. The quality of compression is dependent on the size of the codebook and is evaluated in section 5.2.1.

The actual quantization step is straightforward and requires searching the codebook to find the closest match for each vector in the difference volume. There are several ways of doing so, but in this particular case the simplest possible algorithm turns out to be ideal.

The *full codebook search* algorithm (explained in section 2.4.2), where each input vector is compared to every vector in the codebook, is locally optimal [58] and always results in the best possible match being found. On the other hand, it

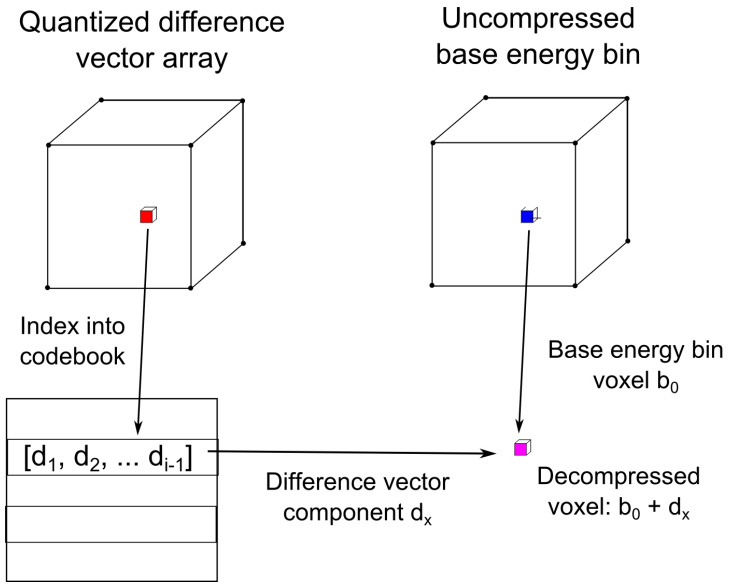

Figure 3.8: Decompression of a voxel from energy bin x of a spectral CT dataset comprising i energy bins. The component of the difference vector that corresponds to energy bin x is retrieved from the codebook and added to the base voxel.

is quite computationally expensive and so faster but less accurate methods such as tree structured VQ [72] have been proposed instead. However, full codebook search is a parallel problem which greatly benefits from GPU acceleration. A CUDA version of this algorithm has been implemented to quantise difference volumes generated by the pre-processing step of VQ1 and has been found to be extremely fast. Quantisation of a difference volume using a pre-generated codebook is shown on the right side of Fig. 3.7.

3.3.3 VQ1: Real-time Decompression on a GPU

This section describes the theory behind decompression of data compressed with VQ1, whereas Chapters 4 and 5 provide benchmarks and performance measurements in real-world situations when VQ1 is integrated into a volume rendering application.

After compression with VQ1, two volume datastes are generated: the base energy bin in an uncompressed form and a volume of indices into the difference vector codebook. This property of VQ1 may provide an advantage in certain situations: the bin that contains the most important data can be chosen as the base to fully preserve its contents from modification.

Decompressing a voxel of a dataset compressed with VQ1 is extremely simple.

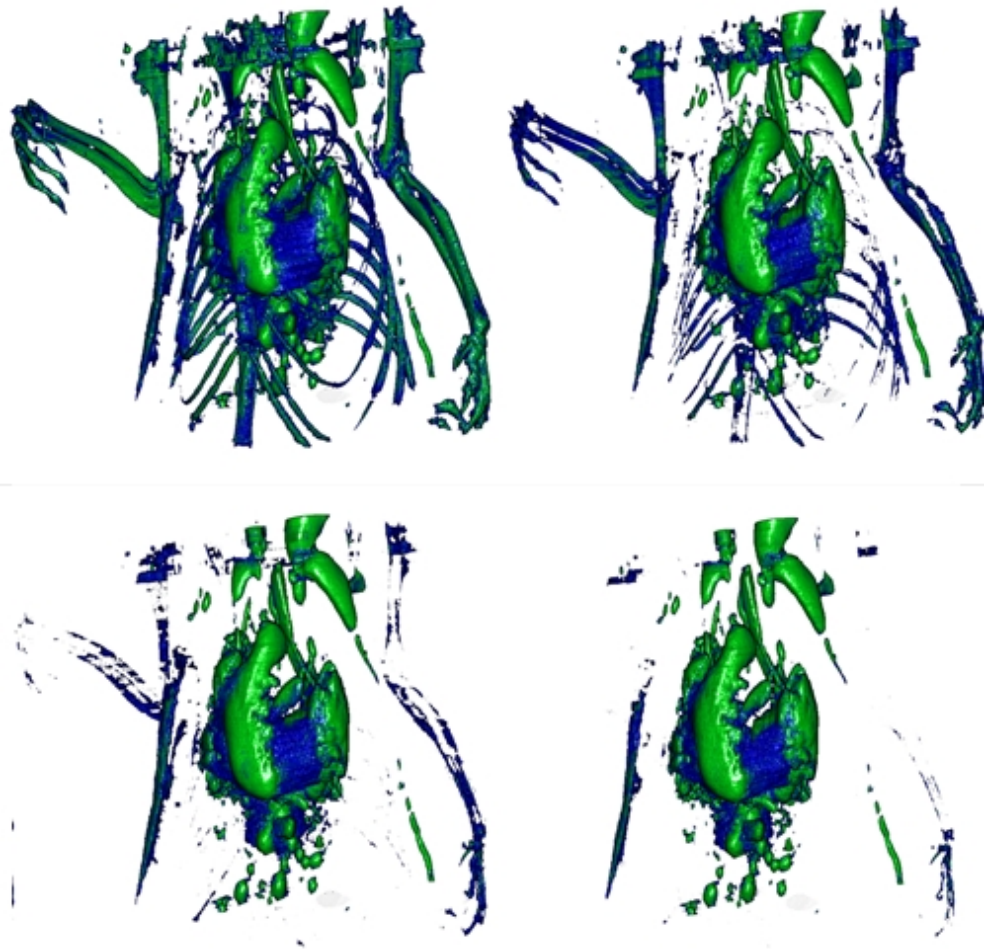

Figure 3.9: 4 energy bins of the Mouse12 dataset compressed with VQ1 and visualised with MARSCUDAVR. Clockwise from top left: 15 keV, 23 keV, 35 keV and 30 keV energy bins.

Obviously, if a voxel belongs to the base energy bin, it does need to be decompressed at all, and the entire process involves a single lookup into the base energy bin array.

If the voxel being decompressed does not belong to the base energy bin, the corresponding index into the difference codebook is also retrieved. A codebook lookup is then performed and the appropriate component of the difference vector is added to the base value (Fig. 3.8). Decompression therefore involves only three accesses to global memory, some of which may be eliminated by caching on the GPU.

Both the base volume and the quantised difference volume are stored separately in arrays inside GPU memory. This ensures that expensive transfers along the PCI-

E bus are avoided and the entire rendering process takes place in-core. The only exchange between the GPU and the CPU/RAM takes place when the rendered frame is copied back to RAM to be displayed using OpenGL commands for drawing buffers.

Fig. 3.9 shows the Mouse12 dataset after being compressed with VQ1 and rendered with MARSCUDAVR. The 15 keV energy bin is the base and has not been compressed, while the other energy bins are rendered directly from compressed spectral CT data. Comparison with the uncompressed Mouse12 dataset is performed in section 5.2.2.

3.3.4 VQ1: Compression Ratio and Efficiency

As this algorithm is developed for compression of 16-bit medical datasets, two 16-bit values are stored per voxel, regardless of the number of energy bins in a dataset: a voxel in the base energy bin and an index into the codebook. While the index itself does not have to be a 16-bit value, in practice a codebook will have more than 256 vectors, and so a data type (such as an *unsigned short* in the *C* programming language) capable of storing larger indices is needed. The size of the codebook is negligible compared to the size of the compressed dataset, so it may be safely ignored when calculating the compression ratio.

Compression ratio varies with dataset size, as it depends entirely on the number of energy bins. An 8-bin dataset will be compressed at 4:1 and this ratio will increase linearly as more bins are added. If extremely large datasets are visualised, this algorithm requires a GPU to have a large amount of graphics memory: for example, for a 16-bit 1024x1024x512 dataset, 2GB is needed to store compressed data. While such GPUs exist (workstation-level GPUs such as the NVIDIA Quadro and Tesla series generally have two or more GB of memory), they are expensive and relatively uncommon. If visualisation of multi-gigabyte spectral CT datasets is performed on hardware with a standard amount of video memory (less than 1.5 GB, for example), a different algorithm may need to be used. Such an algorithm, VQ2, is described in section 3.4.

A somewhat similar compression scheme can be envisaged: it is possible to combine the voxels of all energy bins at each coordinate into a vector and quantise it. This would, in theory, compress the dataset quite well because only a single index into a codebook would be needed to describe all voxels at each coordinate, instead of two values per coordinate as in VQ1.

In practice, however, this method actually results in much poorer visual quality due to the difficulty of finding a good codebook with a small amount of distortion. Generating a well-fitting codebook for a difference volume, on the other hand, is

significantly easier, as difference vectors are much more similar to each other than vectors composed of unprocessed energy bin data.

3.3.5 VQ1: Conclusion

The approach used in VQ1 is based on one specific property of spectral CT datasets that helps preserve image quality: as described in section 3.1.2, the *differences* between voxels in energy bins at the same voxel coordinates are fairly consistent across the dataset. Large numbers of voxels share common sequences and it is this property that can be used to minimise distortion during the generation of codebooks. It should also be noted that VQ1 is applicable not only to spectral CT datasets, but also to any kind of data that has a high degree of correlation between several n-dimensional datasets.

3.4 The VQ2 Algorithm

Along with the VQ1 algorithm, that has been specifically designed to take advantage of unique properties of spectral CT datasets, another algorithm, VQ2, has been created over the course of this research to provide a higher compression ratio at the expense of image quality and decompression speed. It is based on the algorithm presented by Schneider and Westermann [59, 73], which was developed to compress 3D RGB textures by decomposing them into blocks of 4^3 voxels and subsequently compressing each block using a vector quantisation-based approach.

3.4.1 VQ2: Pre-processing

VQ2 is applied separately to every energy bin in a spectral CT dataset. A bin is broken up into blocks of 4^3 voxels and each block is then further decomposed into its mean value and two difference vectors according to the process described below.

First, the mean value of each of the eight blocks of 2^3 voxels within the 4^3 voxel block is found and subtracted from the original data, forming a 64-component difference vector (first row of Fig. 3.10). The same process is repeated with the remaining eight mean values: their mean is found and subtracted from each value, forming an 8-component difference vector (second row of Fig. 3.10).

Thus, the original energy bin is now reduced by a factor of four in every dimension. For every 4^3 voxel block in the bin, there now exists an 8-component and a 64-component difference vector and a mean value. The difference vectors are in fact 2^3 and 4^3 3D volume datasets (as shown in Fig. 3.10) packed into a 1D

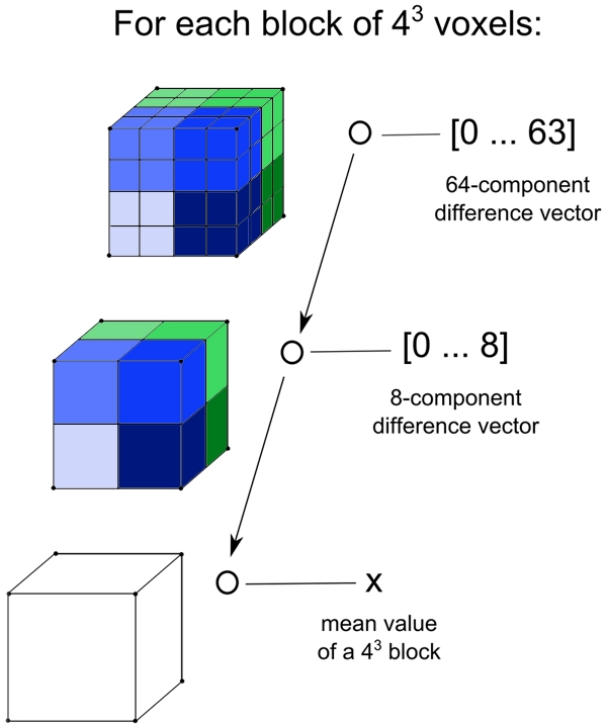

Figure 3.10: Pre-processing of data for the VQ2 algorithm. two difference vectors and a mean value are found per block of 4^3 voxels. Vectors are compressed in a subsequent vector quantisation step.

array. These vectors must now be reduced to indices into a codebook via vector quantization.

3.4.2 VQ2: Compression by Vector Quantization

The process of vector quantisation is exactly the same as in VQ1: training sequences of user-defined size (10% of the dataset by default) are randomly chosen from the full set of difference vectors and separate codebooks for both 8-component and 64-component difference vectors are generated.

These codebooks are then used to quantise the difference vectors according to the full codebook search algorithm. Measurements of the time taken to generate codebooks at various quality settings are presented in section 5.1 and compared to the time taken by VQ1's codebook generation step.

After quantisation of two sets of difference vectors, the following values are created and stored for each 4^3 block of original voxels:

- The mean value of a block.

- An index into the 8-component difference vector codebook.
- An index into the 64-component difference vector codebook.

For each energy bin in a dataset, five arrays are stored:

- Array of mean values of 4^3 voxel blocks (16 bits per value).
- Array of indices into an 8-component difference vector codebook (16 bits per index value).
- Array of indices into a 64-component difference vector codebook (16 bits per index value).
- 8-component difference vector codebook (512-4096 entries).
- 64-component difference vector codebook (512-4096 entries).

VQ2 compresses each energy bin at the ratio of 64:3, as it stores three values per block of 64 voxels. Some additional space will be taken up by the codebooks, but, as with VQ1, it is considered to be negligible compared to the space taken up by the compressed dataset itself. Once all energy bins are processed, compression is complete.

3.4.3 VQ2: Real-time Decompression on a GPU

In order to decompress a single voxel in a 4^3 block, the components of the 8D and 64D vectors that correspond to that voxel's position are added to the mean, reproducing an approximation of the original. Suppose a voxel at coordinates $(403, 413, 102)$ in a volume dataset needs to be decompressed. However, this dataset has been compressed by a factor of four in each dimension, and, therefore, the coordinates must be adjusted.

This address needs to be broken up into coordinates within the downsampled volume and within the local 4^3 block. Therefore, the mean and two codebook indices are retrieved from position:

$$(\lfloor 403/4 \rfloor, \lfloor 413/4 \rfloor, \lfloor 102/4 \rfloor) = (100, 103, 25)$$

of their respective arrays. The local coordinates are:

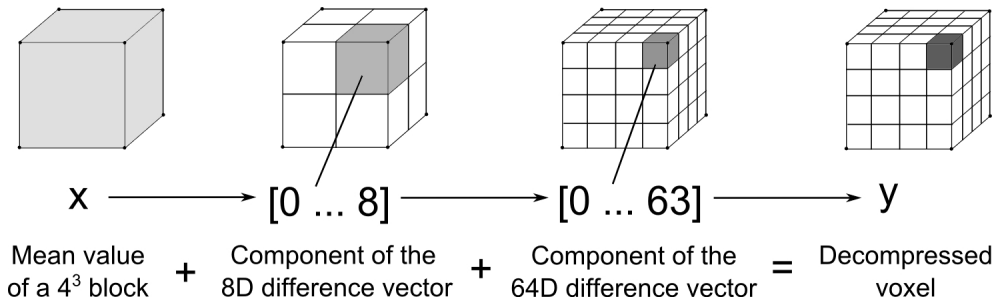

Figure 3.11: Decompression of a voxel using the VQ2 algorithm.

$(403 \bmod 4, 413 \bmod 4, 102 \bmod 4) = (3, 1, 2)$

The component of the 64D difference vector that corresponds to position $(3, 1, 2)$ is retrieved and so is the component of the 8D difference vector that corresponds to position:

$(\lfloor 3/2 \rfloor, \lfloor 1/2 \rfloor, \lfloor 2/2 \rfloor) = (1, 0, 1)$

The two vector components are then added to the mean and decompression is complete.

The process of decompression is shown in figure 3.11. Here, a voxel y (top right of the rightmost volume) is reconstructed by adding together the mean value of a 4^3 voxel block and the corresponding components of the 8D and 64D difference vectors. This figure does not show the extra steps of retrieving the components of the two difference vectors from their respective codebooks, but they do need to be performed in practice.

Overall, this is a much less efficient decompression procedure than that of VQ1, as twice as many retrievals from global memory need to be done per voxel, in addition to the cost of constantly calculating the mapping from global to local coordinates (modulus operations are computationally expensive in CUDA [35, 42]). Since each energy bin in a spectral CT dataset is decompressed individually, there also is a smaller chance of a cache hit and overall cache utilisation is most likely worse than in VQ1.

3.4.4 VQ2: Conclusion

VQ2 illustrates a different approach to spectral CT dataset compression, offering high compression ratios with a corresponding decrease in image quality and decompression speed. It is useful when extremely large datasets need to be visualised

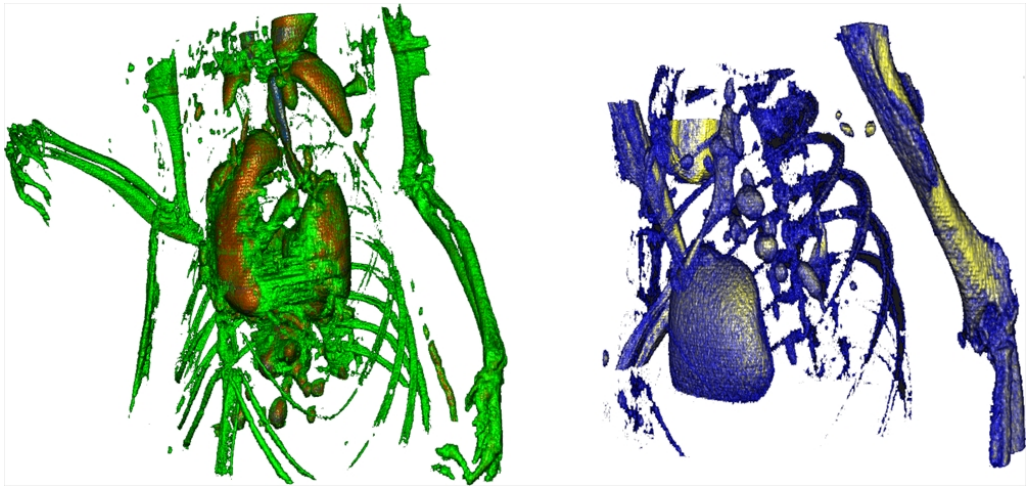

Figure 3.12: Left: the 15 keV energy bin of the Mouse12 dataset compressed with VQ2 and visualised with MARSCUDAVR. Right: 10 keV energy bin of the Mouse6 dataset.

on consumer-grade GPUs that may not have enough memory to hold two large 16-bit volumes, as needed by VQ1.

Block-based compression utilised by VQ2 is not without its problems: there are noticeable visual artifacts such as boundaries between blocks that may be clearly visible in some cases and surfaces that may not appear smooth. Chapter 5 examines the quality loss caused by using VQ1 and VQ2 to compress spectral CT datasets and provides specific suggestions on how to apply VQ2 to minimise visual distortion.

3.5 Summary

This chapter has described the properties of spectral CT datasets and presented two compression algorithms that are suitable for use as part of an interactive visualisation application for spectral CT data. Several important contributions have been made:

- The properties of spectral CT datasets have been examined in this chapter. In particular, exploiting the correlation between the energy bins of a spectral CT dataset has been identified as a potential way to achieve good compression quality.

- Requirements for an asymmetric compression/decompression algorithm suitable for spectral CT datasets have been described.

- VQ1, a novel algorithm, based on inter-energy bin correlation in a spectral CT dataset, has been presented.

- VQ2, an algorithm originally designed for real-time rendering from compressed 3D RGB texture formats, has been adapted for use with spectral CT datasets.

Chapter 4 will show that, in practice, both VQ1 and VQ2 algorithms are easily integrated into a volume rendering application for spectral CT datasets and allow for interactive frame rates during visualisation. Chapter 5 will demonstrate that image quality is not severely degraded if spectral CT data is processed and visualised with these algorithms.

Chapter IV

Visualisation of Compressed Spectral CT Datasets

This chapter discusses some aspects of visualising compressed spectral CT datasets on modern GPU hardware using MARSCUDAVR, a custom application designed for this purpose over the course of this project.

First, the hardware used for development and testing of VQ1, VQ2 and MARSCUDAVR is described in section 4.1. Next, the high-level design of MARSCUDAVR is introduced in section 4.2. The practical challenges involved in rendering directly from compressed spectral CT data formats using the CUDA technology are examined in section 4.3. Techniques that may be used to accelerate visualisation of spectral CT data are presented in section 4.4 and possible post-processing effects are discussed in section 4.5.

4.1 Hardware and Software

Two computer systems have been used for development, testing and evaluation of compression and visualisation algorithms described in this thesis: one based on dual modern gaming-grade NVIDIA GeForce GTX470 graphics cards (Compute Capability 2.0) and one based on the older NVIDIA Quadro FX5800 workstation-level graphics card (Compute Capability 1.3) [35].

Table 4.1: Specifications of GPUs used for testing MARSCUDAVR.

Specification	**Quadro FX5800** secondary testing system	**GeForce GTX470** primary testing system
CUDA Compute Capability	1.3	2.0
CUDA Cores	240	448
Memory (MB)	4096	1280
Shader Clock (MHz)	1100	1215
Core Clock (MHz)	650	607
Memory Clock (MHz)	1632	1674
Memory Bandwidth (GB/s)	102	133.9

The primary development and testing system had an Intel Core i7 870 CPU with hyperthreading turned on, 8 GB of DDR3 RAM in dual-channel configuration and two GTX470 GPUs. The machine was running the Windows 7 operating system.

The secondary testing system was based on the Intel Core i7 920 CPU with hyperthreading turned on, 12GB of RAM in triple-channel configuration, one Quadro FX5800 GPU and was also running the Windows 7 operating system.

The two systems are similar enough to make the differences in CPU and RAM speeds irrelevant. For all intents and purposes, only the GPUs affect the execution speed of compression and visualisation algorithms presented in this thesis.

The majority of testing and evaluation was performed on a single GTX470 graphics card and, therefore, all frame rates reported in this chapter and Chapter 5 were measured for this GPU, unless stated otherwise.

4.2 Application Overview

MARSCUDAVR is the volume rendering application developed over the course of this research for the purpose of visualising large compressed spectral CT datasets at interactive frame rates. This section is intended to serve as a high-level overview of the software and explain the general concepts behind MARSCUDAVR. Specific algorithms and visualisation techniques are covered in later sections.

MARSCUDAVR combines compression of spectral CT data (as part of a preprocessing chain) with visualisation, with the two parts interacting through the use of common data formats. This allows for greater extensibility - that is, compression algorithms and visualisation techniques are mostly separate and independent of each other, but can be combined in different ways depending on the problem. Additional functionality can also be introduced at later stages depending on the needs of the MARS project.

This application has been designed as a proof-of-concept and is not intended to be a standalone visualisation package for spectral CT data. It is hoped that in the future MARSCUDAVR will be integrated with MARSCTExplorer, which already has a custom UI well suited to the task [19].

In MARSCUDAVR, the technically challenging and slow compression process has been separated into its own application that is run independently of the visualisation software. This is done because compression and decompression should be asymmetrical in order to support real-time rendering from a compressed format (section 3.2). In other words, it should be significantly easier and faster to decompress data than to compress it.

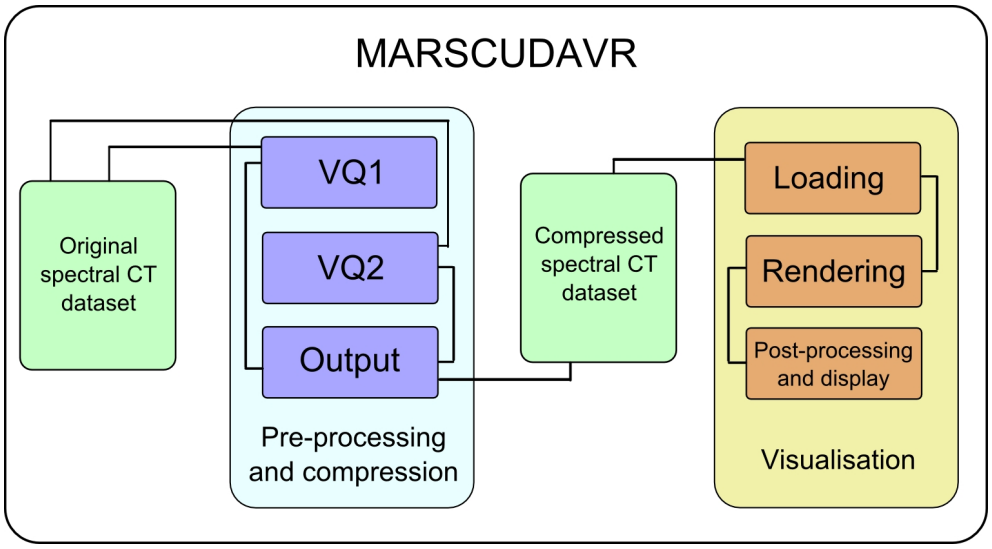

Figure 4.1: Structure of MARSCUDAVR. The pre-processing part of the package can compress a spectral CT dataset with either VQ1, VQ2, or with both algorithms and then output compressed data to disk. The visualisation part loads compressed data and displays it, adding post-processing effects, if necessary.

MARSCUDAVR can compress spectral CT datasets with VQ1 and VQ2, as described in Chapter 3. Once a spectral CT dataset is compressed, it is written to disk in a custom binary format and can then be visualised as many times as necessary without going through the compression process again. Datasets that are too large to hold in RAM (for example, over 2-3 GB in size) are first split into smaller subsets that are individually processed with VQ1 or VQ2. These subsets are then combined into a full compressed volume dataset and written to disk.

The visualisation part of MARSCUDAVR is a volume raycaster that is implemented in C++ and CUDA. It reads compressed datasets from disk and uploads them directly to the GPU without performing any further processing. The rendering code is completely independent of the algorithm used for spectral CT data compression. The only routine that is algorithm-specific is the decompression function that provides raw voxel data when requested by the raycaster. This function is changed depending on whether datasets compressed with VQ1 or VQ2 are being visualised. Post-processing operations can also be applied to the resulting images. Fig. 4.1 shows the structure of MARSCUDAVR and the way spectral CT data normally gets processed and visualised.

As multi-gigabyte datasets are common in spectral CT, the entire visualisation toolchain that starts with data pre-processing and ends with rendering is accelerated through the use of CUDA. This introduces significant benefits, as most

data processing operations that need to be performed on spectral CT datasets are parallelisable.

For example, vector quantization is a parallel task that is perfectly suited for execution on GPUs. Codebook generation, however, is a sequential process which is left to the general-purpose CPU that is well-adapted to such computation. Overall, heterogeneous programming is the key to optimising the application as a whole.

4.3 Real-time Rendering of Compressed Spectral CT Datasets with CUDA

Chapter 3 has described in detail the process of transforming a spectral CT dataset into more compact formats that can be uploaded to a GPU and visualised at interactive frame rates. The theory behind non-real-time compression and real-time decompression of spectral CT datasets using two different algorithms based on vector quantisation has also been explained.

The actual implementation of these algorithms on a GPU as part of a volume rendering application remains a complicated practical matter. Several aspects of this problem are covered in this section, which is concerned with some of the technical challenges involved in visualising compressed spectral CT datasets using GPU hardware and the CUDA programming language.

Rendering from compressed formats generated by VQ1 and VQ2 is relatively easy and fast compared to alternative schemes. The majority of non-trivial and sequential work is done during the pre-processing stage, leaving the GPU to perform simple arithmetic operations and easily parallelisable retrievals from global memory.

4.3.1 Volume Raycasting with CUDA

As stated above, the visualisation algorithm used in MARSCUDAVR is called volume raycasting, which has been explained in section 2.2.2.1. The basic version of this algorithm can be easily implemented in CUDA: each ray can be processed by a single CUDA thread. Branching is fully supported, so a thread can operate independently, vary the number of samples it takes, change lighting algorithms if needed, and so on. This, however, may come at some cost to performance.

The most significant change from traditional volume raycasting affects data storage. Texture memory (areas of GPU video memory used for the purposes of storing texture data) used to be a natural storage place for volume data (for example, [74, 24]), as shader-based volume raycasting techniques could only access

this type of memory and a GPU's built-in trilinear interpolation hardware could be used to efficiently calculate the sample value at each sampling point.

Compressed spectral CT datasets may also be stored as textures inside GPU memory, but this confers no advantage, as hardware texture interpolation is inapplicable to compressed formats (see section 4.3.2 below). Furthermore, new generations of NVIDIA's graphics cards introduce larger and faster L1 and L2 caches [35] that can be used to cache any data retrieved from global memory. Therefore, MARSCUDAVR will not benefit from the performance improvements that were previously offered by using texture memory for data storage.

In MARSCUDAVR, all spectral CT data is packed into large 1D arrays in global GPU memory, while textures are only used to store transfer functions. In order to access a voxel based on its 3D coordinates, an index into the packed 1D array needs to be generated. The following formula is used:

$$i = p_x \cdot vd_y \cdot vd_z + p_y \cdot vd_z + p_z \tag{4.1}$$

where p is a coordinate of a voxel in 3D space and vd describes the dimensions of the 3D volume dataset.

When a voxel is accessed, decompression functions specific to VQ1 or VQ2 are executed, depending on the chosen compression method. The voxel is decompressed according to the algorithm described in section 3.3.3 for VQ1 or section 3.4.3 for VQ2. Even though voxels are not accessed directly, all algorithms related to visualisation (such as those responsible for interpolation and gradient estimation at sampling points) operate as though they are, having no knowledge of the actual compressed data formats or the work that needs to be performed to decompress voxel data.

In conclusion, MARSCUDAVR implements a standard volume raycasting algorithm: rays are cast through the volume dataset using the front-to-back traversal scheme, taking samples at regular intervals and gathering colour and opacity, as determined by a transfer function. The only difference is that compressed spectral CT data is stored in packed 1D arrays instead of the standard method of storing uncompressed volume data in 3D textures.

4.3.2 Interpolation

MARSCUDAVR implements the basic raycasting algorithm, but some modifications need to be made in order to account for the specifics of the problem of visualising compressed spectral CT datasets using CUDA. The first of these changes relates to interpolation.

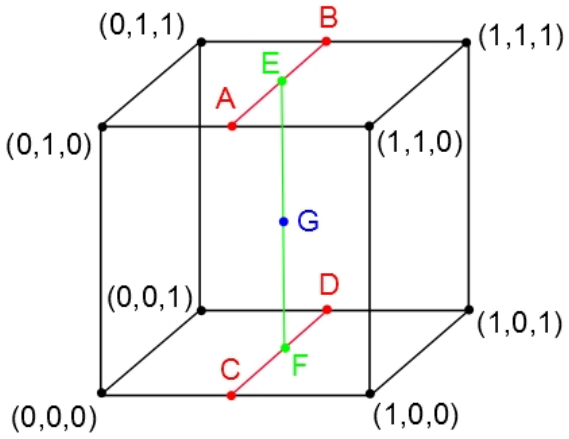

Figure 4.2: Trilinear interpolation from eight nearest voxels to calculate sample value at point G.

The interpolation technique used in MARSCUDAVR is called trilinear interpolation. This algorithm is straightforward and involves retrieving eight voxels that are nearest to the sampling point and performing seven linear interpolations, as shown in Fig. 4.2. For example, point A is found by linearly interpolating between $(0, 1, 0)$ and $(1, 1, 0)$, while B is found by linearly interpolating between $(0, 1, 1)$ and $(1, 1, 1)$. E is then found by linearly interpolating between A and B and so on. Finally, the value at the sampling point (G) is calculated.

Most previous attempts at volume raycasting (including MARSCTExplorer for spectral CT data visualisation [19]) have uploaded uncompressed volume data to the GPU as a 3D texture, which, during sampling, has made it possible to take advantage of specialised trilinear interpolation hardware that is an essential feature of all modern graphics cards. In MARSCUDAVR, spectral CT data is stored in a compressed format, which means that hardware interpolation units cannot be used: indices into codebooks are discrete values that are completely independent of each other and specialised retrieval algorithms are required.

This problem has been previously identified by Schneider and Westermann [59], but the hardware limitations of their day meant that they could only implement nearest neighbour interpolation in their volume rendering application. MARSCUDAVR has the advantage of running on newer, more powerful GPUs that allow a software trilinear interpolation scheme to be used. Manual retrieval, decompression and interpolation of the sample value from eight nearest voxels is, as expected, a very slow process. Fermi-series graphics cards, for example, have four texture

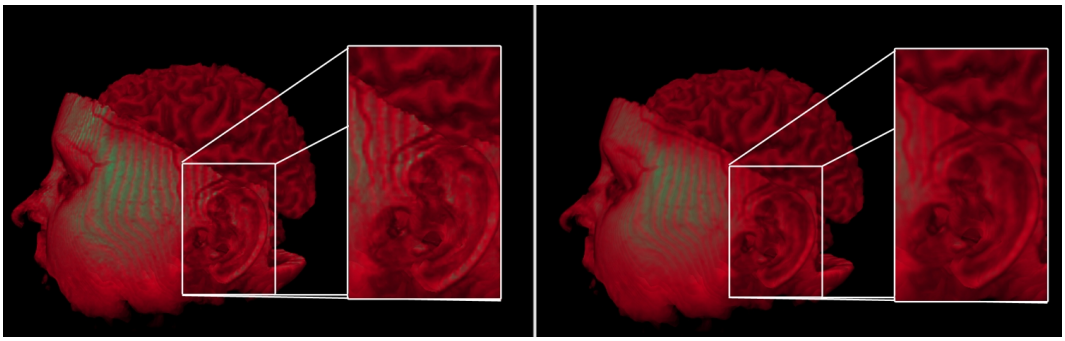

Figure 4.3: The MRBrain dataset rendered with a modified version of the volume raycaster provided as part of the CUDA 4.0 SDK. Left: trilinear interpolation. Right: cubic B-spline interpolation.

units per multiprocessor that have been specifically designed for fast texture fetching and interpolation [37]. These units cannot be used for sampling of volume data in MARSCUDAVR and, consequently, a significant performance drop is observed, as demonstrated in section 5.3.

Ideally, a different interpolation scheme should be used, such as interpolation based on cubic B-splines, as proposed by Ruijters et al. [75]. A CUDA version of tricubic interpolation for 3D textures has been implemented for testing purposes, based on *Cg* [76] code provided in Ruijters' paper. NVIDIA's implementation of a volume raycaster that is included in the CUDA 4.0 SDK [46] has been modified to use spline-based interpolation, as opposed to the default trilinear interpolation method. Results for the MRBrain dataset can be seen in Fig. 4.3.

Clearly, cubic B-spline interpolation is superior to trilinear interpolation. Detail is preserved, yet surfaces appear smoother and wood-grain artifacts (artifacts that look like growth rings on a piece of wood, resulting from undersampling [73]) are less prominent. Integration of this interpolation technique into MARSCUDAVR is not possible, however, as this code only works with 3D textures and an implementation that operates on CUDA arrays would cause frame rates to drop down to non-interactive levels.

4.3.3 Isosurface Extraction

Isosurface finding methods improve the quality of images rendered by volume raycasting algorithms by applying certain criteria to volume data and extracting a distinct boundary between regions of interest and the surrounding empty space. Some methods, like the well-known Marching Cubes algorithm [20], generate a 3D mesh of the isosurface that can be visualised separately from the volumetric

```
1  if ( Current sample is inside region of interest )
2  {
3      if ( Number of valid samples = 0 )
4      {
5          IsoPrevious = Previous sample value - threshold
6
7          IsoCurrent = IsoPrevious * ( Current sample value - threshold )
8
9          for ( C = 0.5f; C >= 0.0015625f; C *= 0.5f,
10             IsoCurrent = IsoPrevious * ( Current sample value - threshold ) )
11         {
12             if (IsoCurrent < 0.0f)
13                 Sampling location -= Ray direction * Step size * C
14             else
15                 Sampling location  += Ray direction * Step size * C
16
17             Take sample
18         }
19     }
20 }
```

Figure 4.4: Pseudocode for finding the precise location of the isosurface in MARSCUDAVR.

dataset.

Improvements in rendering speed by a factor of 10 have been reported when only the isosurface is visualised [77]. However this visualisation technique may lead to important features being missed, as no data past the isosurface is shown to the user.

It is possible to combine direct volume rendering with real-time isosurface extraction. The isosurface extraction algorithm implemented in MARSCUDAVR dynamically adjusts the sampling position to find the exact location of the boundary between empty space and the object of interest. When the isosurface is found, the ray does not terminate, but instead carries on until other termination conditions are triggered. Therefore, enhanced surface detail and features inside the volume dataset can be shown at once with no quality loss. In MARSCUDAVR, isosurface extraction is used purely to improve visual quality, and is, in fact, slightly detrimental to the frame rate of the application.

Pseudocode for finding the exact location of the isosurface is shown in Fig. 4.5. When a ray begins sampling inside a non-empty region for the first time (conditions on lines 1 and 3), this code is executed. Essentially, the sampling position is moved back and forth along the sampling ray and the distance is halved with each iteration (*C *= 0.5f* on line 9). If the sampling position has moved beyond the isosurface back into empty space, the direction is reversed and if hasn't yet reached the isosurface, the direction does not change (lines 12-15). Fig. 4.5 illustrates the

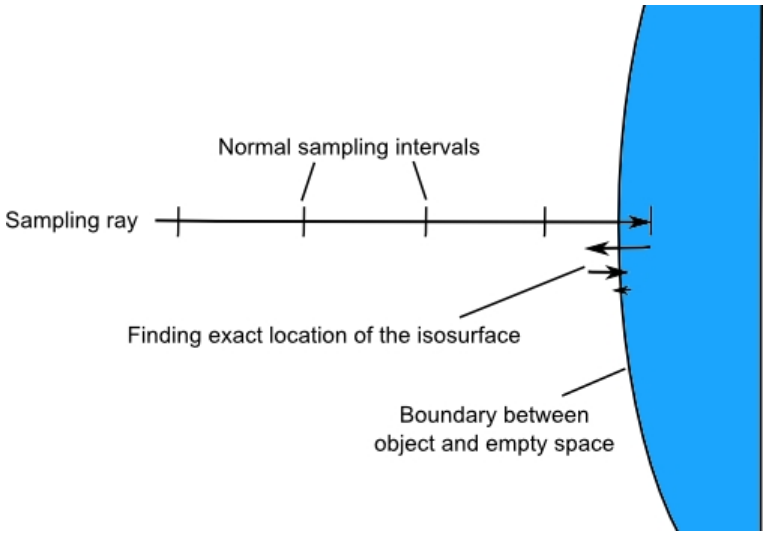

Figure 4.5: Isosurface finding method used in MARSCUDAVR. When the first sample inside the region of interest is taken, the ray is moved back and the sampling position is adjusted until the exact location of the isosurface is found.

same process graphically.

4.3.4 Volume Illumination

Volume illumination generally involves approximating light transfer within a volumetric dataset to enhance visible geometric detail. Normally, a gradient is calculated at each sampling point and used to modify the sample value based on the view direction according to a shading model [73]. In volume rendering, the camera is normally considered to be a light source, so, for example, the surfaces facing away from it would be darker, while the surfaces facing the camera would have specular reflections. As illustrated by Fig. 4.6, more subtle detail can be seen and the dataset representation appears more realistic.

MARSCUDAVR uses two methods to calculate the gradient at a sampling point. Both involve computing it in real-time, as pre-calculated gradients increase the space needed to store the volume dataset by a factor of four [33].

The first method is tied to the software trilinear interpolation scheme described in the previous section. Here, the gradient at a sampling point is calculated at the same time as the interpolated sample value itself, based on the eight nearest neighbours. Since interpolation needs to be performed in any case, computing the gradient requires no further accesses to global memory and only consists of arithmetic operations on data in fast register memory.

The gradient is formed by subtracting neighbouring values to find the gradients

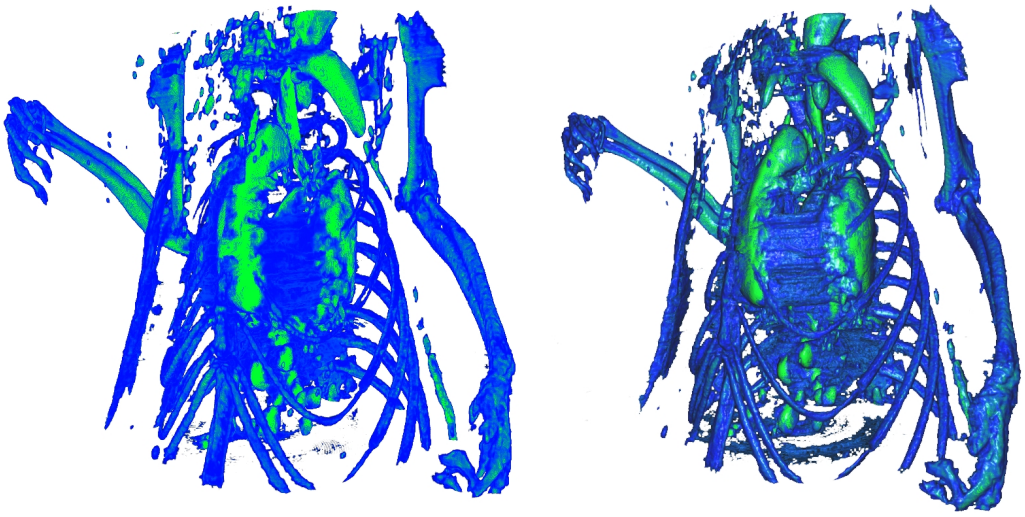

Figure 4.6: Volume lighting as implemented in MARSCUDAVR. Left: no lighting. Right: lighting based on the Phong-Blinn shading model with the gradients estimated using central differences.

in the x, y and z directions. The gradients are then weighted according to the coordinates of the sampling point. Appendix B contains the CUDA code used for this interpolation scheme.

Overall, this method is very fast, but only generates crude approximations of the real gradient at a sampling position. It results in blocky images and, in MARSCUDAVR, is only used during rotation, where high image quality is of lesser importance than execution speed.

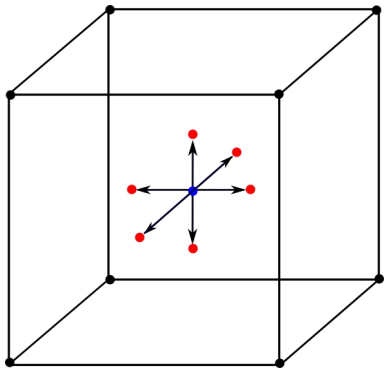

Figure 4.7: Calculating the gradient at a sampling point in MARSCUDAVR using central differences. Gradient at the sampling point (blue) is found by combining the sample values at the six points marked in red.

The second method is called central differences [23], where the sampling position is moved one unit back and forward along the x, y and z axes. Figure 4.7 illustrates the approach, showing the six extra samples that are taken to calculate the gradient at the actual sampling point. Sample values are subtracted from one another to find the gradient in each direction, according to the algorithm given in Appendix B.

As expected, taking seven samples per position instead of one significantly affects the frame rate of MARSCUDAVR. Section 4.4.3 examines the frame rates with and without the central differences gradient estimation method.

Once the gradient is found by either method, the Blinn-Phong shading model [78] is applied. This model is easily adaptable to GPU architectures and numerous implementations have been proposed for various purposes and light types [76]. The dot product of the gradient vector and the view direction is calculated and used to find the diffuse component, which is in turn used to derive the specular component. The interpolated sample value is multiplied by the diffuse component and the specular component is added to the result.

4.4 Acceleration and Optimisation

3D volume renderers place high demands on the hardware they run on. Over the past decades, technological progress has been steady, offering faster hardware at more affordable prices. However, the increasing sizes of volume datasets and the desire for high-quality near-photorealistic rendering still drive the need for research into acceleration techniques for 3D volume rendering.

Research into acceleration is particularly relevant to this project because the performance of MARSCUDAVR is severely degraded by the need to decompress spectral CT data and by an inefficient interpolation scheme. As this section will demonstrate, if certain acceleration and optimisation techniques are applied, performance can be increased to the point where interactive visualisation is possible.

4.4.1 Measuring Performance

As explained above, MARSCUDAVR supports real-time decompression and rendering from spectral CT data that has been compressed by two different algorithms: VQ1 and VQ2. However, decompression using VQ1 is more computationally efficient (section 3.3.3), and frame rates will be higher than if VQ2 was used. Therefore, all performance measurements presented in this thesis will specify which algorithm is being used to decompress data during rendering. In addition, the frame rate of MARSCUDAVR is also influenced by the number of samples per

ray, gradient estimation algorithm, transfer function, volume opacity, acceleration methods and so on.

Due to these factors, it is very difficult to give an average frame rate for MARSCUDAVR, so subsequent sections will only attempt to examine specific cases, such as assessing the frame rate of MARSCUDAVR using VQ1 and VQ2 for a particular dataset with other parameters being equal, or comparing the frame rate at different sampling rates with all other parameters fixed.

4.4.2 Early Ray Termination

The most obvious acceleration method is early ray termination, where rays cast through the volume dataset are stopped when they either leave the volume bounds or if the opacity of the accumulated sum over a ray reaches a predefined threshold (such as 95 or 99%). Early ray termination generally results in a significant performance increase [24], as most rays will either reach full opacity before taking the maximum allowed number of samples or will simply exit the volume dataset. Early ray termination has been integrated into MARSCUDAVR: a test is performed after each sample to check for both termination conditions.

Alongside early ray termination, a number of other acceleration techniques have been considered and implemented.

4.4.3 Sampling Rate

MARSCUDAVR reduces the sampling rate by a factor of four when the camera is being manipulated (for example, rotated or zoomed in or out) by the user and increases it when the view is stationary. Using a smaller number of samples during interaction does not affect the perception of data by a human observer in a significant manner (less detail is seen during motion [79]), but it allows the software to maintain an interactive frame rate [80]. In addition to reducing the sampling rate, the standard central differences algorithm used for lighting is also changed to a simpler one described in section 4.3.4.

Two quality settings have been used to test the performance of MARSCUDAVR:

- **High settings** - maximum of 2000 samples per ray, with the central differences algorithm being used to estimate the gradient at every sampling point.

- **Low settings** - maximum of 500 samples per ray with a fast, low precision gradient estimation scheme.

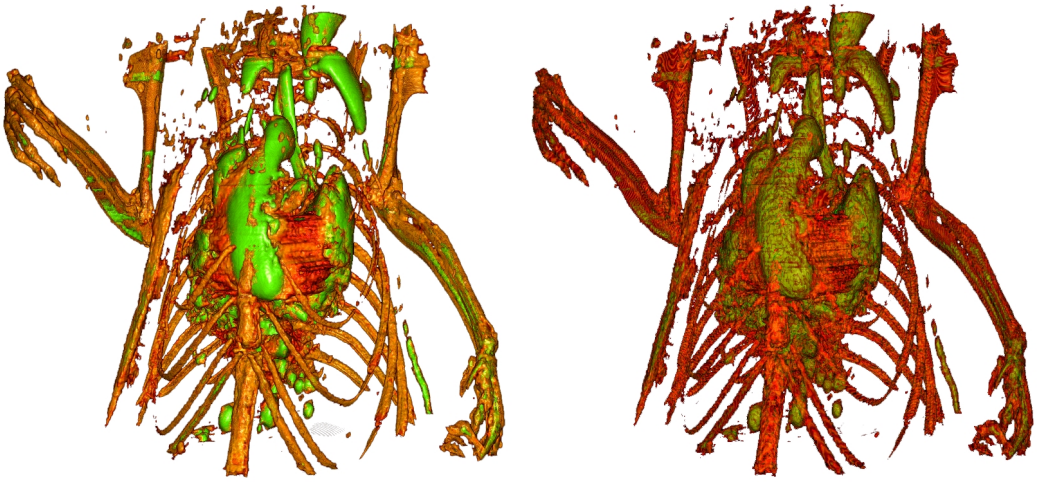

Figure 4.8: The Mouse12 dataset compressed using VQ1 and rendered using MARSCUDAVR. Left: the dataset as it appears when the view is stationary, rendered at high quality settings. Right: the same dataset, as it appears during user interaction (model rotation) at low quality settings.

When the Mouse12 dataset is compressed with VQ1 and visualised with MARSCUDAVR, the following results are observed: the frame rate is 3.74 frames per second (fps) at high settings and 8.59 fps at low settings.

The visual difference between the two quality settings is shown in Fig. 4.8. While there are significant differences between the two images, overall the most prominent geometric features are preserved and the quality is sufficient for the user to continue exploring the dataset. Once the user stops rotating the camera, the sampling rate is increased and the gradient estimation algorithm is changed back to a high-quality central differences scheme.

4.4.4 Min-max Octree and Empty Space Skipping

In general, regardless of the specifics of the volume raycasting algorithm, a spatial data structure can be employed to support and accelerate rendering. In the case of MARSCTExplorer, a normal render target might be a frame of around 1000x1000 pixels, which must be updated in real-time as the user interacts with the visualisation (for example, rotates, zooms in and out or changes settings like lighting or transparency).

Therefore, a million rays may need to be cast multiple times per second, but, due to the decrease in performance caused by decompression of voxel data and subsequent manual trilinear interpolation, this would normally bring the frame rate of MARSCUDAVR down to non-interactive levels.

In some scenes, only 0.2-4% of all samples may contribute to the final rendered image that the user sees [74], and therefore empty space skipping combined with early ray termination can be used to greatly reduce the number of samples and increase performance.

Empty space skipping, as the name implies, is based on avoiding sampling in those regions of the volume dataset that have been classified as empty by a transfer function or by thresholding raw sample values. If a sampling point falls into an empty region, the ray simply advances forward. An auxiliary data structure that is perhaps no larger than $^1/_{32}$ - $^1/_{64}$ of the total size of the original volume dataset is usually employed to classify regions and enable quick skipping of those sampling points that lie inside empty space.

This structure, which may, for example, be an octree [81] or a k-d tree [82], contains certain values that provide information as to how a particular region of the volume dataset should be treated (that is, whether it should be skipped, sampled at a lower rate and so on). The performance increase will generally be so significant that any reduction in memory available for storing the actual volume data can be tolerated.

An octree can be constructed in a number of ways and two are demonstrated in Fig. 4.9. On the left is an example of an octree as described by, for example, Lefebre et al. [83] or Taminen and Samet [84]. A volume dataset is iteratively decomposed into cubic blocks, with the process stopping when a block of space is

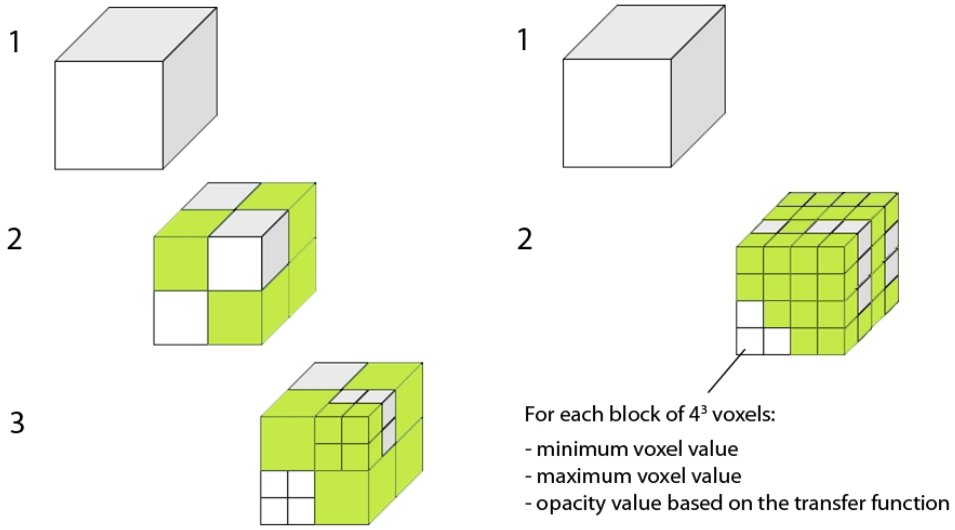

Figure 4.9: Differences between a theoretical octree (left) and the octree implemented in MARSCUDAVR (right).

classified as homogeneous according to certain conditions that vary based on the application.

This process will result in blocks of uneven size, as shown on the left of Fig. 4.9, which is not a problem if the tree is traversed from its root node. However, the constraints imposed by the target GPU architectures and the requirements of this particular project mean that it is better to generate an octree-like data structure that is in fact just a scaled-down representation of the original volume dataset.

As explained in section 2.3.1, in CUDA, access to global memory is significantly slower than operations on registers and shared memory. Traversing an octree such as the one shown on the left of Fig. 4.9 may involve several accesses to global memory, whereas a lookup in an octree shown on the right only involves calculating the index and retrieving the corresponding value from the octree. While the octree on the right does not decompose a volume dataset as efficiently as the one on the left, in practice higher access speeds will compensate for a minor increase in storage requirements.

The octree implemented in MARSCUDAVR follows this principle: each node describes a block of 4^3 voxels and contains the minimum and maximum raw voxel values in this particular block and the average opacity of the block based on the transfer function. The octree only needs to be created once and is independent of the compression method, as it is constructed in a pre-processing step based on the original spectral CT dataset. This process is parallel and fully CUDA-accelerated, which means that creating an octree takes no longer than a few minutes for even the largest spectral CT datasets.

This octree can be utilised in one of two ways: first of all, if the maximum raw voxel value in an octree cell is smaller than a user-defined threshold, then the cell will be classified as empty. No sampling will be done if a ray enters this cell, and the ray will advance forward. Secondly, blocks can also be skipped based on their average opacity, as determined by a transfer function. This approach is based on the work by Li et al. [85] and Kainz et al. [33].

Table 4.2: Performance of MARSCUDAVR with and without using octree-based empty space skipping (ESS). Average frame rates for the Mouse12 dataset compressed with VQ1 are measured.

Quality settings	**With ESS**	**Without ESS**	**Performance improvement (%)**
High quality	3.74 fps	0.49 fps	663%
Low quality	8.59 fps	1.75 fps	391%

Due to the nature of spectral CT datasets, techniques based on assigning colour and opacity to samples by intermixing spectral CT data may yield better visual results than simple classification by the transfer function [19]. Nevertheless, a transfer function can be used. MARSCUDAVR allows the user to choose whether to use thresholding based on raw sample data or on data after classification by the transfer function.

As shown in table 4.2, the use of empty space skipping results in a significant performance improvement, which, for VQ1, is in fact slightly better than the figures reported by Li et al. [85] where the authors expect an increase in frame rate to be around 200-500% if a spatial data structure is used to implement empty space skipping in a volume rendering application.

4.4.4.1 Adaptive Sampling

An octree can also be extended to support adaptive sampling, which, essentially, involves classifying regions of a volume dataset according to a certain metric which assigns "importance" ratings to blocks of space [86]. Areas deemed important can be sampled at a higher rate and those that are of lesser interest can be sampled at a lower rate. In theory, this should reduce the overall number of samples while maintaining a high sampling rate in certain important regions. Gains of 25% while maintaining similar visual quality have been reported [87].

Classification presents the greatest challenge, while actually varying the sampling rate during rendering is reasonably easy. An "importance" value is added to each octree cell which is then used to adjust the sampling rate when a ray is sampling inside it. Due to the difficulty of choosing a meaningful metric, this technique has not been implemented in MARSCUDAVR. However, should a metric be found, adaptive sampling can be easily introduced at a later stage, as the framework to support it already exists.

4.4.5 Empty Space Leaping

Empty space leaping is a technique that involves skipping all empty space between the volume bounds and the region of interest in one step and starting sampling from the point where a ray first takes a non-zero sample value. This can be achieved in several ways, for example by exploiting frame-to-frame coherence and performing reprojection (recalculating new sampling coordinates when the view direction changes) [88, 89].

MARSCUDAVR implements a custom empty space leaping scheme (illustrated in Fig. 4.10), which involves storing the point along each ray where the first non-

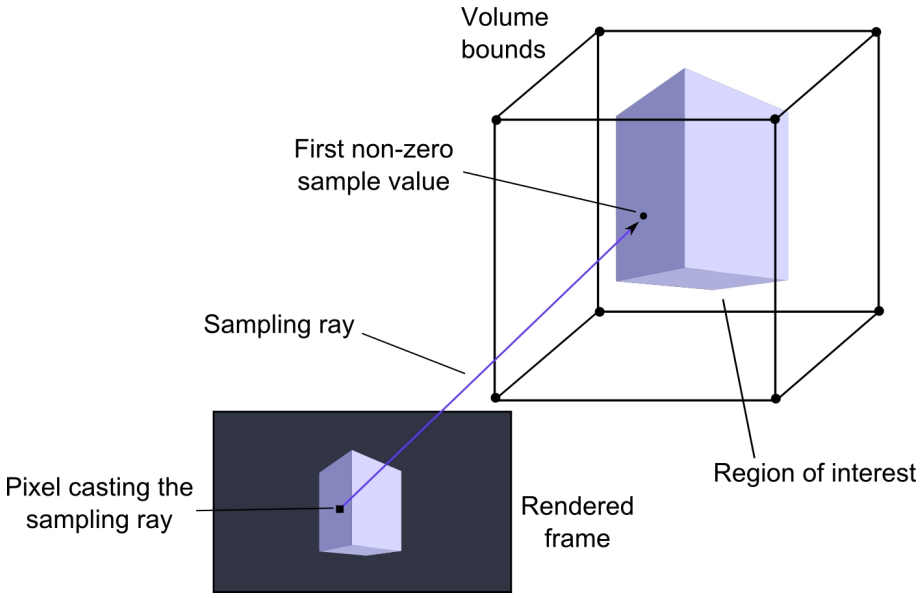

Figure 4.10: Empty space leaping as implemented in MARSCUDAVR. For each ray, the coordinates of the first point at which the sample value is non-zero are stored. In subsequent frames, rays begin sampling from those coordinates, completely avoiding sampling within regions of empty space.

zero sample has been taken. The next time a frame is rendered, rays begin sampling from these points, instantly leaping from the volume boundary to the region of interest. This approach, however, only works when the view is stationary, as it is entirely dependent on the view direction remaining constant.

Testing could detect no difference in performance between pure empty space skipping and a combination of skipping and leaping. This is most likely due to the fact that lookups in the octree are so quick to perform that any difference that exists is so minor that it can't be detected by reasonably precise frame rate measurement (accurate to 0.1 frames per second). This technique, however, may still be useful when certain scenes are being rendered and therefore empty space leaping has been introduced into MARSCUDAVR.

There is also an additional unforeseen benefit of empty space leaping - storing the points where sampling rays first make contact with the region of interest essentially builds up a depth buffer. This leads to several interesting possibilities, such as displaying the depth buffer alone, or using it to modify the rendered image in some way. A post-processing effect based on the depth buffer is described in section 4.5.

Table 4.3: Comparison of frame rates of MARSCUDAVR rendering using a single GPU and dual GPUs. Average frame rates for the Mouse12 dataset at high quality settings are shown.

Graphics Hardware	**VQ1**	**VQ2**
Single NVIDIA GTX470 GPU	3.74 fps	1.58 fps
Dual NVIDIA GXT470 GPUs	6.53 fps	2.81 fps
Performance Improvement (%)	74.6%	77.8%

4.4.6 Rendering on Multiple GPUs

Volume raycasting is a parallel problem and therefore this algorithm should scale perfectly with the number of available processing units. Adapting it to work with an arbitrary number of CUDA devices in a single system is an obvious and highly beneficial extension: it is cheaper to employ a number of low-cost GPUs and split the workload among them than to use a single, powerful, high-end graphics card.

The workload splitting algorithm implemented in MARSCUDAVR is called split-frame rendering. Two or more GPUs render parts of the frame, which is split horizontally, and the results are merged into a single frame. The resulting image is then displayed using standard OpenGL buffer drawing functions. This algorithm is also used by, for example, NVIDIA's SLI (Scalable Link Interface) technology [90].

Table 4.3 shows how using a second GPU increases the frame rate of MARSCUDAVR almost two-fold. In practice, the split-frame rendering algorithm does not scale perfectly because of the need to spawn multiple CPU threads to control the GPU's and to distribute the workload. Merging two separate buffers into a single one and displaying it also causes a small decrease in performance.

The workload can be distributed unevenly during split-frame rendering: one GPU may receive a much more difficult part of the volume dataset to render, while another may have to render mostly empty space. No image will be displayed until all GPUs complete their parts of the frame. Therefore, a serious workload imbalance needs to be avoided, as it brings the overall performance closer to that of a single GPU and negates any benefits offered by split-frame rendering.

An example of even and uneven workloads that can occur as a result of split-frame rendering is shown in Fig. 4.11. To avoid situations like these, a load balancing algorithm has been implemented that dynamically evaluates the time taken to render parts of a frame on each GPU and redistributes the workload as needed. Load balancing ensures that there is seldom a situation where one GPU

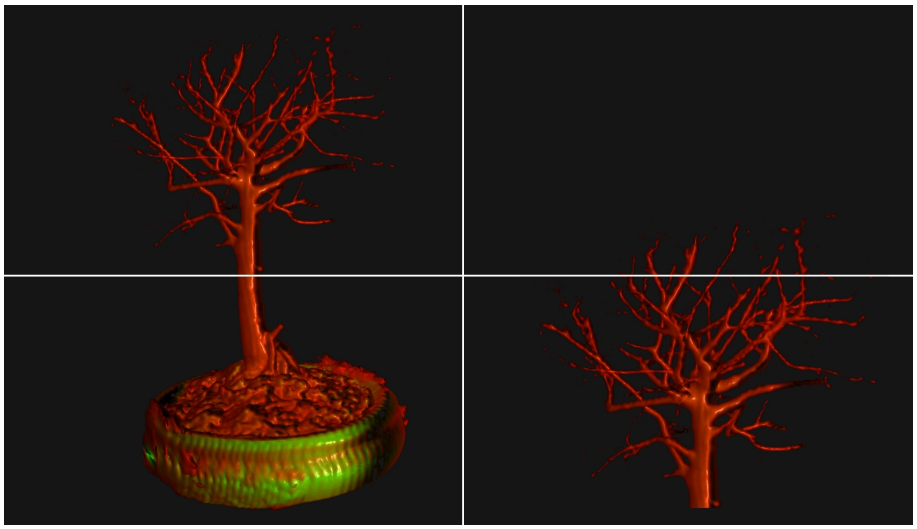

Figure 4.11: Rendering on two GPUs with a naive workload distribution algorithm: both GPUs render 50% of the frame. In some situations (left), it is close to the optimal solution; in others (right), the workloads assigned to the GPUs are uneven and the frame rate may drop substantially.

performs more work than the other and allows a more or less constant frame rate to be maintained.

Table 4.4 shows the difference in frame rate resulting from the use of a load balancing algorithm when rendering on two GPUs. The performance of MARSCUDAVR with VQ1 was increased by 17% when load balancing was used and the performance of MARSCUDAVR with VQ2 was increased by 18.9%.

These results, however, should be interpreted with caution. The improvement in speed resulting from the use of load balancing algorithms is impossible to predict with certainty, as it depends entirely on the parameters being used for rendering and the composition of the scene being visualised.

Table 4.4: Performance of MARSCUDAVR on two GPUs with load balancing on and off at low quality settings. Average frame rates measured for the Mouse12 dataset are shown.

Setting	**VQ1**	**VQ2**
Load balancing off	11.79 fps	9.8 fps
Load balancing on	13.80 fps	11.65 fps
Performance improvement (%)	17%	18.9%

4.4.7 CUDA-specific Optimisation

This section discusses certain CUDA and GPGPU-specific optimisations that could be applied to MARSCUDAVR's volume raycasting algorithm in order to increase its execution speed. These techniques are complementary to the acceleration methods described above and merely involve tuning code for faster execution on NVIDIA's Fermi GPU architecture.

4.4.7.1 L1 Cache and Shared Memory

An important feature of the CUDA programming language is the ability to vary the amounts of Level 1 (L1) cache and shared memory (SM) available to the multiprocessors in a CUDA device. For devices based on the Fermi architecture, out of a common block of 64 KB of memory per multiprocessor, 48 KB can be dedicated to either L1 or SM. Each option has certain consequences on performance based on the nature of the given computational task and the structure of the CUDA kernel being executed [35]. Both options have been tested for VQ1 and VQ2 and results are presented in table 4.5.

Clearly, having a larger amount of L1 cache is better for both algorithms. The reason is that shared memory is only used to store a few rarely-accessed variables. Therefore, it makes no difference whether the kernel can access 16 KB or 48 KB of shared memory per multiprocessor.

Having a larger cache, on the other hand, results in a 29% increase in rendering speed for MARSCUDAVR using VQ1 and 34.9% increase for VQ2 because the frequency of accesses to global memory is reduced. While some kernels may benefit from storing more data in shared memory, this particular implementation of the volume raycasting algorithm clearly favours larger cache sizes.

Table 4.5: Effect of different amounts of L1 cache and shared memory on the performance of MARSCUDAVR using VQ1 and VQ2 at low quality settings.

Configuration	**VQ1**	**VQ2**
16 KB L1 Cache and 48 KB Shared Memory	6.73 fps	5.53 fps
48 KB L1 Cache and 16 KB Shared Memory	8.78 fps	7.46 fps
Performance improvement (%)	23.4%	25.87%

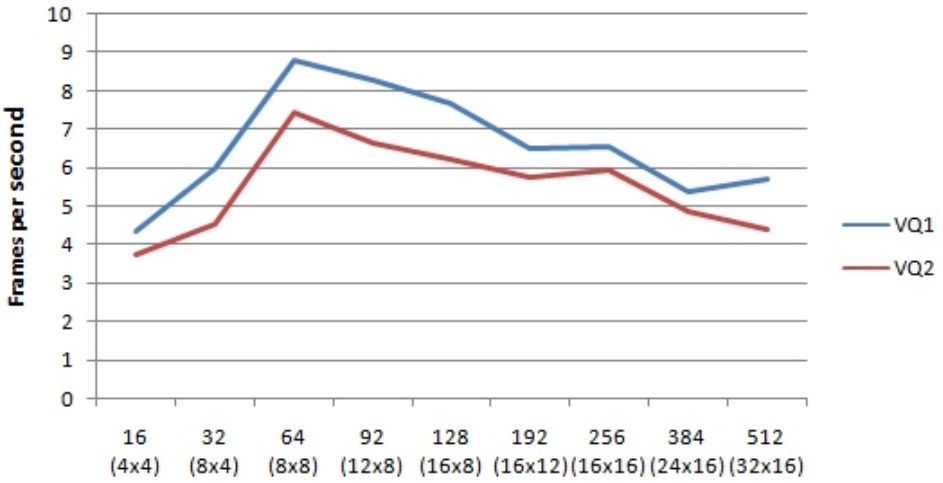

Figure 4.12: Relationship between CUDA block size and frame rate of MARSCUDAVR using VQ1 and VQ2, as measured for the Mouse12 dataset at low quality settings.

4.4.7.2 Block Size

Determining the optimal block size for a CUDA kernel is one of the most important tasks for a CUDA programmer [42]. There is no set formula to calculate block size; its dimensions should be determined experimentally. Several factors affect the choice of block size: kernel complexity, frequency of accesses to global memory, register usage and device architecture.

Fig. 4.12 illustrates the relationship between block size and execution speed. 64 threads per block turns out to be the optimal value. The number of threads per block that gives the best performance for each algorithm is in agreement with the advice given by NVIDIA: due to the uncertainty introduced by the factors mentioned above, it is best to experiment with a number of combinations.

4.4.7.3 Memory Access Patterns

The memory access patterns that yield the highest performance are aligned and strided access patterns [42]. This holds true for all generations of NVIDIA's graphics cards, but newer devices of Compute Capability 2.0 (such as the NVIDIA GTX470 used during this project) are much more tolerant of misaligned memory accesses due to improved caching and architectures specifically optimised for GPGPU tasks [35].

Sampling during volume raycasting is performed on a more or less random basis: the sampling point of one ray may be in a completely different position in

relation to the sampling point of another ray. This results in unpredictable accesses to locations in global memory and, therefore, memory access patterns cannot be optimised in order to increase the execution speed of MARSCUDAVR's volume raycasting kernel.

It is also not possible to either optimise the layout of data in global memory or to change memory access patterns to improve performance without resorting to complex multi-pass rendering schemes or delegating some pre-processing work before each frame to the CPU. It is also questionable whether implementing these complex algorithms will even lead to any increases in rendering speed. Simply accepting the performance penalty for misaligned memory access may result in higher frame rates.

Therefore, it has been decided not to attempt any optimisation of memory access patterns for either VQ1 or VQ2. However, for more complex non-real-time rendering algorithms that aim to provide higher image quality it may be a good idea to consider optimisation by pre-calculating, for example, which memory blocks will be accessed by which rays and adjusting the rendering algorithm accordingly.

4.4.7.4 Optimisation of Voxel Retrieval and Interpolation Methods

Voxel retrieval and interpolation methods need to be adapted for execution on GPU architectures just like any other functions used by MARSCUDAVR's volume raycasting code. Optimisation of these algorithms is crucial, as they provide raw voxel data to all subsequent functions, such as those responsible for lighting calculation and compositing of samples.

When visualising real spectral CT datasets, the user may decide to combine a number of energy bins in an arbitrary fashion [19]. This means that any voxel may be re-used several times during a single intermixing step. Therefore, retrieving voxels when they are needed and discarding them afterwards may become a highly inefficient access method. Consider the following intermixing algorithm that combines data from three energy bins:

$$\text{Sample} = (Bin_0 - Bin_1) + (Bin_0 - Bin_2) \tag{4.2}$$

If a naïve algorithm is used, then for each sample a voxel from energy bin 0 will need to be retrieved twice. In practice, it would be faster to retrieve the data from all energy bins for a given voxel coordinate once and place these values into fast register-based storage. While some voxels may never be used during the intermixing stage, no further accesses to slow global memory will be needed. This is an example of a modification that needs to be made to algorithms because of

constraints imposed by GPU architectures and the specifics of the problem being solved.

4.5 Post-processing

This section discusses the possible use of image-space post-processing techniques in order to enhance the clarity of the final image or generate extra effects that may be difficult to add during the rendering pass. In MARSCUDAVR, post-processing can be performed in real-time through the use of a second CUDA kernel (or a series of kernels) launched immediately after the rendering kernel completes and generates a frame.

The techniques destribed in this section can be applied regardless of compression or volume data formats, as they only operate on the final 2D image. Standard image processing algorithms can be applied to these images to increase contrast, substitute colours, emphasise edges and so on.

One important benefit of utilising CUDA to implement volume rendering algorithms is the ability to easily generate and randomly access any number of buffers within GPU code (limited, of course, by the amount of video memory on the GPU). This allows for a variety of effects to be applied to the rendered image at little cost, as only small buffers need to be created and image processing techniques are by nature highly parallelisable. In addition, some operations, such as Gaussian blur, cannot be applied during the rendering pass.

One example of a post-processing effect that may be used to enhance certain features of an image is the Sobel operator. It is a kernel applied to a pixel and the eight pixels in its immediate neighbourhood in the x and y directions. If A is taken to be a region of 3x3 pixels in an image centered on pixel (x, y), the filter can be thought of as two convolutions, as shown below:

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} * A \quad \text{and} \quad G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} * A \tag{4.3}$$

Gradient magnitude G at (x, y) can then be found using G_x and G_y:

$$G = \sqrt{{G_x}^2 + {G_y}^2} \tag{4.4}$$

Gradient magnitude can be used for edge detection, as areas with the largest gradient are likely to appear as edges to a human eye. This approximation may not be entirely accurate, but, as shown in Fig. 4.13, it may still be useful for practical purposes, as most real edges (such as the boundaries between, for example, bones

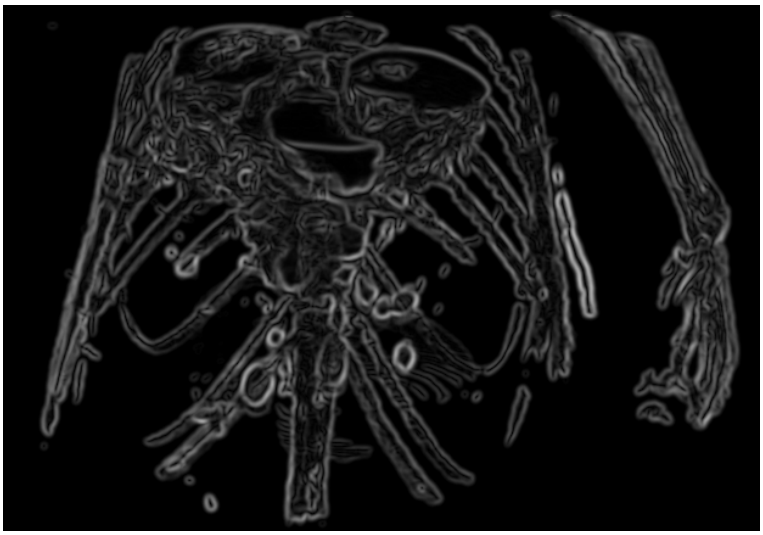

Figure 4.13: A rendering of 230 slices (out of 512) of the Mouse12 dataset processed with a Sobel operator for edge extraction. The outlines of the skeleton and the internal organs are clearly visible.

and empty space) are extracted from the image.

However, gradient magnitude can also be used to modify the original image to emphasise features such as edges. Consider the case of applying the Sobel operator to an image with full volume illumination (for example, to a rendering of the Mouse12 dataset shown on the right of Fig. 4.6 or on the left of Fig. 4.8). Highlights and highly reflective areas may be picked up as edges, which is clearly not correct. However, with CUDA, the following sequence of actions can be performed:

- During the rendering pass, render the image as normal, with full volume lighting, but store the specular value at each pixel in a separate array instead of adding it to the final pixel value.

- During the first round of post-processing, apply Gaussian blur to the rendered image to smooth it and ensure that only thick, prominent edges are detected.

- During the second round of post-processing, apply the Sobel operator to the blurred image. This results in a new image that shows the magnitude of the gradient at every pixel (Fig. 4.13).

- Blend this image with the original rendered frame (Fig. 4.14).

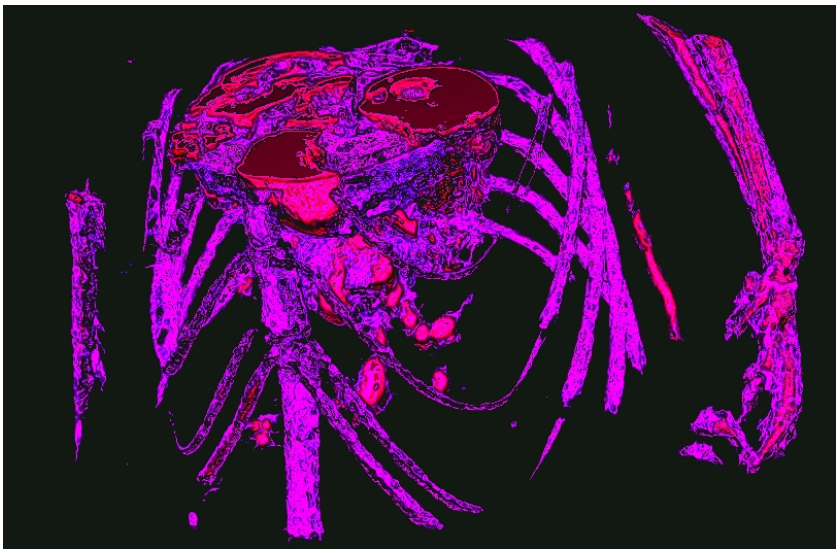

Figure 4.14: Image obtained as a result of blending a rendering of the Mouse12 dataset (using a transfer function and high quality volume illumination) with edges extracted by the Sobel operator (shown in Fig. 4.13). Minor surface details and some edges are highlighted.

- Optionally, add specular values to each pixel.

This process results in an image where specular reflections are not incorrectly picked up as edges and may serve to sharpen some boundaries and bring out small details and features. These effects can be beneficial to the end-user as they may lead to improved analysis of the dataset.

Depth of field is another example of a post-processing effect that can be easily introduced into MARSCUDAVR. Empty space leaping (section 4.4.5), records the coordinates at which the first non-zero sample of each ray was taken and stores them in a separate buffer. This information can then be used during post-processing to selectively apply a Gaussian blur function to some regions of pixels that are, for example, far away from the camera, enhancing the perceived depth of a 3D scene. Such depth cues may be used in conjunction with those already provided by volume illumination (see section 4.3.4).

Figures 4.15 and 4.16 illustrate this technique, which can be a valuable addition to a future high-quality photorealistic renderer for spectral CT datasets.

In conclusion, MARSCUDAVR allows multiple kernels to conduct several rendering and post-processing passes and can use a large number of buffers to store auxiliary information. While this project does not aim to identify and implement useful post-processing functions, it is nevertheless important to note the range of

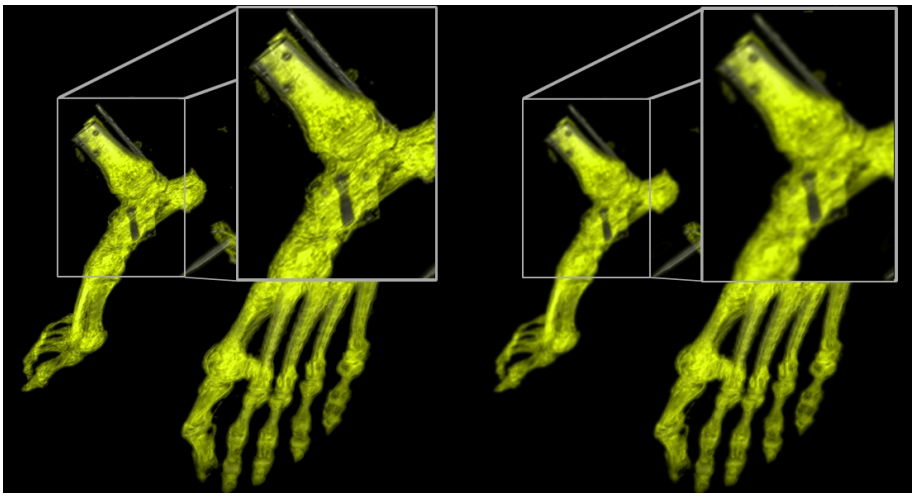

Figure 4.15: Depth of field as a post-processing effect applied to an image of the Vakhum6 Mummy dataset (http://www.ulb.ac.be/project/vakhum/) rendered with MARSCUDAVR. Left: no depth of field. Right: basic depth of field effect implemented by blurring pixels based on the distance from camera.

their potential applications. Post-processing, in the form described in this section, can be introduced into any CUDA-based volume rendering system.

It has been previously found that adding typical post-processing effects (smoothing, sharpening or feature enhancement) as part of direct volume rendering may result in better visual quality than using image-based enhancement approaches [91], as information about the geometry of a volume dataset is only available during the rendering pass. Such information can be used to highlight edges or modify lighting effects without distorting colour. This is noted as being a possible direction for further research, as spectral CT datasets contain a large number of features that can be be used to generate enahancement effects during rendering.

4.6 Summary

This chapter has shown that it is possible to create a real-time interactive visualisation system for compressed spectral CT datasets. Modern GPUs are highly programmable and fully capable of supporting not only basic raycasting, but also auxiliary techniques for improving image quality and frame rate. In order to implement this functionality, existing visualisation methods have been adapted and new compression schemes have been designed. In particular, this chapter has shown that:

- Interactive visualisation of compressed spectral CT datasets is possible on a

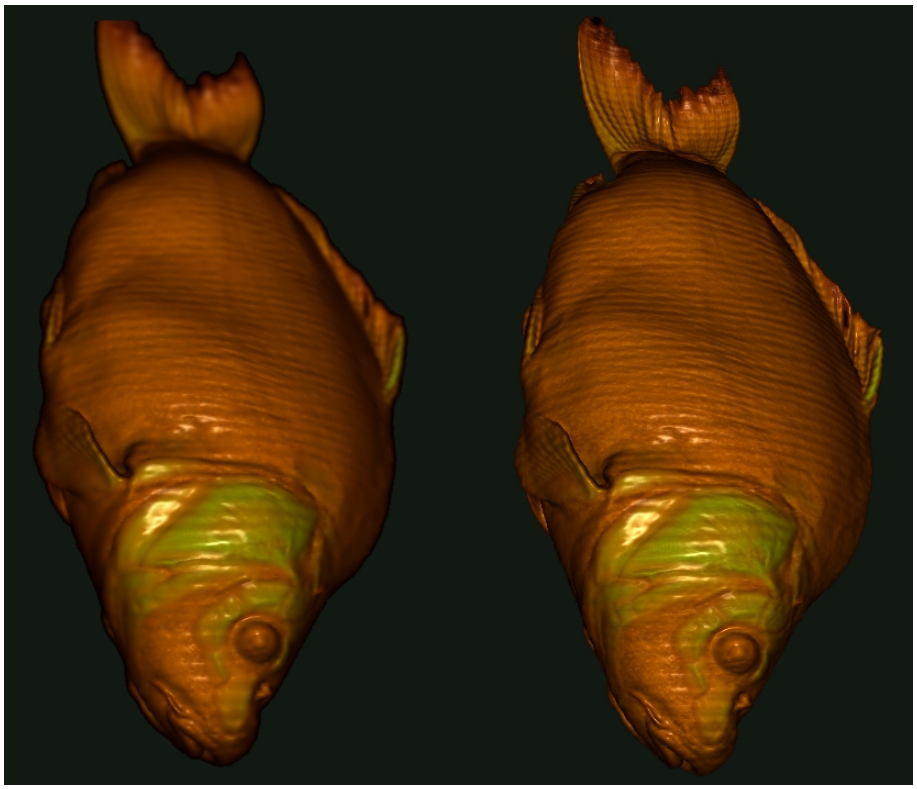

Figure 4.16: Left: depth of field effect applied to a rendering of the Carp dataset based on distance information gathered during the rendering pass. Right: no post-processing.

single high-end consumer-grade GPU. Frame rates of around 9 fps for VQ1 and around 7 fps for VQ2 are possible at low quality settings.

- By dynamically varying the sampling rate and changing the algorithm used for gradient estimation it is possible to maintain interactive frame rates when the user is interacting with volume data and provide high visual quality when the view is stationary.

- Existing acceleration techniques such as early ray termination and empty space skipping can be used to accelerate visualisation of spectral CT data. Octree-based empty space skipping, in particular, improves the frame rate of MARSCUDAVR by over 400%.

- The speed of spectral CT data visualisation can also be improved by distributing rendering across an arbitrary number of GPUs using the split-frame rendering algorithm with load balancing.

- Optimisation is critical for extracting the most out of available hardware resources, as GPUs are very sensitive to poorly written and unoptimised code.

- Post-processing can be used to enhance images generated by the main rendering algorithm. Effects such as blurring, sharpening, gradient magnitude extraction and depth of field can be implemented at little cost to performance.

Chapter V

Evaluation

This chapter evaluates the quality of images produced by MARSCUDAVR and investigates the distortion and quality loss caused by compressing spectral CT datasets with VQ1 and VQ2. First of all, the time taken to generate codebooks for both algorithms at different settings is calculated and the relationship between codebook quality and generation speed is discussed in section 5.1.

The visual quality offered by VQ1 and VQ2 is evaluated in section 5.2. Three spectral CT datasets are studied in this section and conclusions are drawn about the correct way to apply each compression scheme. The effects of noise on compression and visualisation of spectral CT datasets are also discussed in this section.

The frame rate and visual quality of MARSCUDAVR is compared to MARSCTExplorer in section 5.3. Finally, the influence of GPU architecture on the execution speed of MARSCUDAVR is discussed in section 5.4.

5.1 Codebook Generation Times

This section compares the codebook generation steps of VQ1 and VQ2 to evaluate how long it would take to process a spectral CT dataset at average quality settings (1024 vector codebook for VQ1 and dual 1024 vector codebooks for VQ2) and how codebook generation time scales with increasing dataset size.

5.1.1 VQ1

For VQ1, the generation time for a given codebook size is a function of both vector length and size of the training sequence. A few examples are shown in Figures 5.1 and 5.2. Here, the Phantom dataset ($280 \cdot 280 \cdot 200$ voxels per bin, eight energy bins) is processed with VQ1 and codebooks are generated. As mentioned in section 3.3.2, the training sequence is chosen randomly, which results in varying generation times, as it will be easier to find well-fitting codebooks for some training sequences and harder for others. Therefore, the codebook generation times measured in this section are an average of five separate attempts.

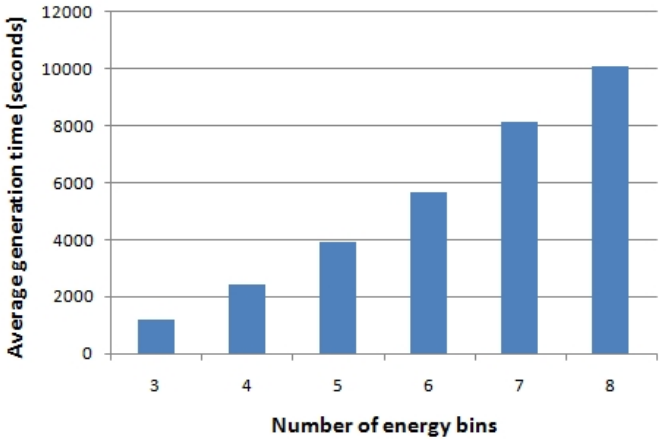

Figure 5.1: Time taken to generate 1024-vector VQ1 codebooks for the Phantom dataset with respect to the number of energy bins.

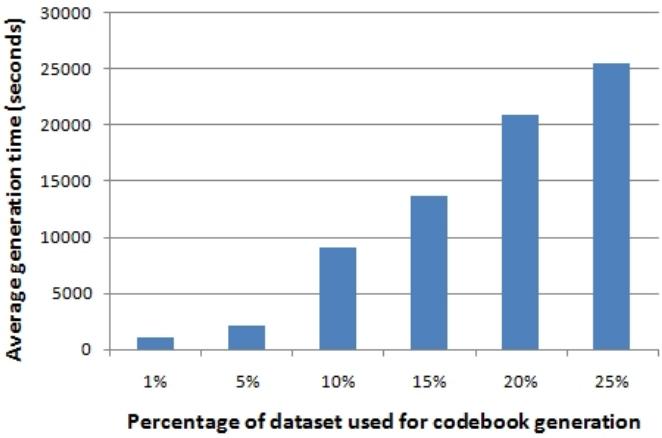

Figure 5.2: Time taken to generate 1024-vector VQ1 codebooks for the Phantom dataset with respect to training set size. The size of the training set is shown as a percentage of the original dataset size.

Figure 5.1 shows that the time taken to generate a VQ1 codebook varies with respect to the number of energy bins in a dataset. The percentage of the Phantom dataset used as a training sequence is fixed at the default value of 10% and a progressively larger number of energy bins is processed, starting with three (a base bin and 2-component difference vector volume) and finishing with the full dataset of eight energy bins (a base bin and 7-component difference vector volume). Generation time scales linearly with regard to the number of energy bins, which means that processing datasets consisting of a large number of energy bins with VQ1 is feasible.

Figure 5.2 shows how processing times vary with respect to the size of the training set used during codebook generation. The number of energy bins is fixed at 8, resulting in the largest possible difference vector length. At the default setting of 10%, the average generation time is just over 9000 seconds, or over 2.5 hours. This is acceptable, since a codebook only needs to be generated once per dataset. However, if required, a smaller percentage of the dataset can be used as a training sequence.

5.1.2 VQ2

The process of generating codebooks for VQ2 differs from VQ1 for three reasons. First of all, two codebooks (for 8 and 64-dimensional vectors) need to be generated for each energy bin in a spectral CT dataset. VQ1, on the other hand, requires a single codebook for the entire dataset.

Second, each energy bin is processed separately and therefore codebook generation can be parallelised. However, in order to ensure a fair comparison to VQ1, all energy bins have been processed sequentially and the total time has been measured.

Finally, in VQ2, each energy bin is reduced in size by a factor of four in all three dimensions (section 3.4). Therefore, the size of the new volume dataset, and consequently, the number of vectors that can be used to form a training sequence

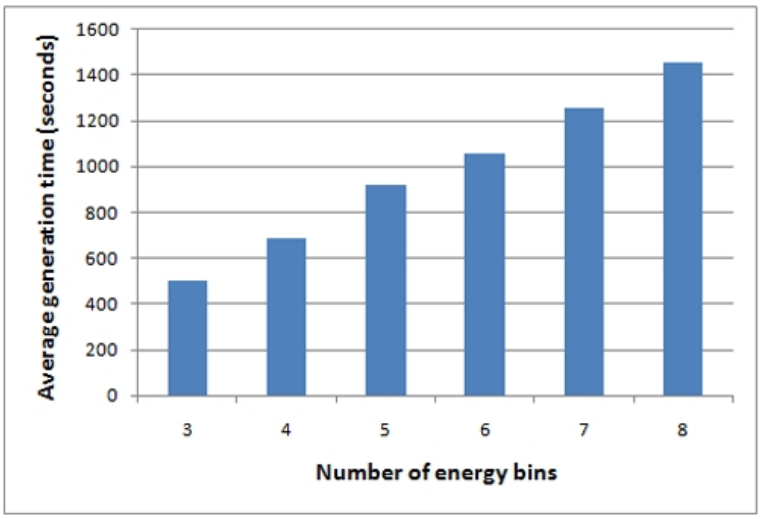

Figure 5.3: Codebook generation times for the Phantom dataset processed with VQ2. The percentage of the original dataset chosen as a training sequence is fixed at the default value of 10% and the number of energy bins for which codebooks are generated varies between three and eight.

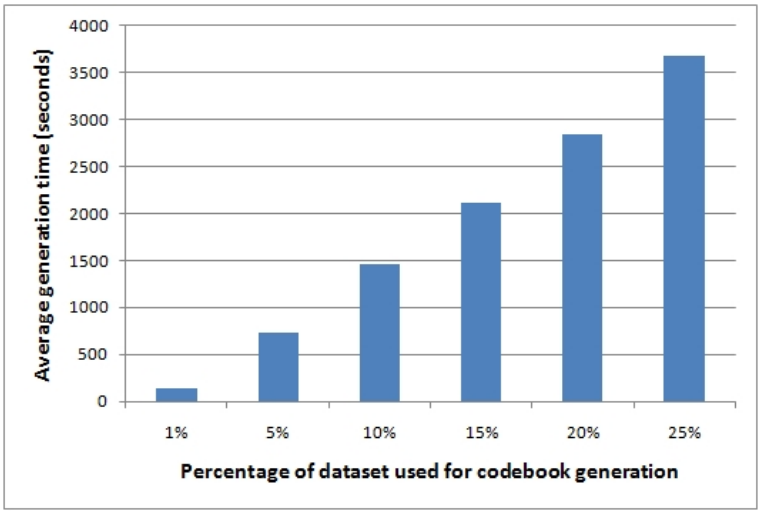

Figure 5.4: Codebook generation times for the Phantom dataset processed with VQ2. The number of energy bins is fixed at eight and the percentage of the original dataset chosen as a training sequence is varied between 1% and 25%.

is reduced by a factor of 64, but the vectors themselves are longer than those normally expected in VQ1.

Fig. 5.3 shows the relationship between the number of energy bins in a dataset and codebook generation time. The training set size has been fixed at 10%, just as with VQ1. As expected, the time increases linearly because each bin is treated separately.

Fig. 5.4 shows how the percentage of the dataset randomly chosen to serve as a training sequence affects the time taken by the codebook generation step of VQ2. Here, the number of energy bins for which codebooks are generated is fixed at eight, but the percentage of the original dataset used as a training sequence varies between 1% and 25%. As with VQ1, the increase in time is linear.

5.1.3 Conclusion

As shown above, VQ2 has a clear advantage over VQ1 in terms of the time taken to generate codebooks for a given spectral CT dataset. However, VQ2 utilises block-based compression and is inherently less visually appealing than VQ1 (see section 3.4 and specifically section 3.4.4). It is therefore possible to attempt to generate larger codebooks (for instance, of 4096 vectors or more) in an attempt to enhance visual quality. This will likely take the same amount of time as generating an average-sized codebook for VQ1.

5.2 Image Quality

Evaluating the quality of compressed data is a complex topic: there exists no standard evaluation method and user expectations vary depending on the data type being studied. In this particular case, the problem is exacerbated by the novelty of spectral CT technology. Numerous studies have examined the effects of compression on medical images 2.5, but the quality of compressed spectral CT data remains unexplored. Furthermore, even the properties of *uncompressed* spectral CT datasets and their effects on visualisation have not been studied in great depth.

Therefore, a reasonably conservative approach to evaluation has been taken. It consists of calculating PSNR (method described in section 2.4) for several compressed spectral CT datasets and performing visual comparisons of images rendered using MARSCUDAVR.

5.2.1 Codebook Size and Visual Quality

VQ1 and VQ2 are both based on vector quantization, which means that two factors affect the quality of compression: the codebook generation algorithm and codebook size. In this case only a single generation algorithm is used, and therefore the size of the codebook is the sole factor that has a direct effect on the distortion caused by compression. PSNR metrics for several codebook sizes along with the corresponding rendering speeds are displayed in table 5.1.

The table shows that larger codebooks always result in better visual quality. This is to be expected, since quality is quite clearly dependent on how well a codebook describes the original set of uncompressed vectors and larger codebooks will provide closer-matching options.

Table 5.1: Effect of codebook size on the PSNR of VQ1 and VQ2 for the Mouse12 dataset (average PSNR of all energy bins).

Codebook size (number of vectors)	**VQ1 PSNR (dB)**	**VQ2 PSNR (dB)**
512	30.7	27.1
1024	30.9	27.6
2048	31.2	27.8
4096	31.3	31.65

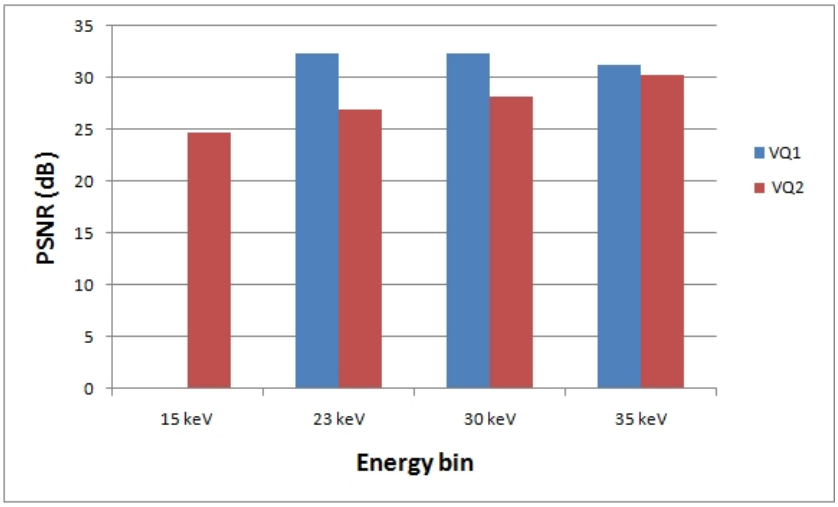

Figure 5.5: PSNR of the Mouse12 dataset compressed with VQ1 and VQ2. The 15 keV energy bin has been chosen as a base for VQ1 and thus has not been compressed.

5.2.2 Comparison of Three Spectral CT Datasets

Three spectral CT datasets (Mouse12, FatCaFe and Phantom) have been processed with VQ1 and VQ2 and visualised with MARSCUDAVR. These datasets, described in detail in their respective sections, illustrate how the use of compression affects visual quality and how dataset size and noise influence compression and visualisation.

5.2.2.1 Mouse12 Dataset

This dataset is 4GB in size (four energy bins of 1024x1024x512 voxels), but it is not the largest dataset that the MARS-CT scanner can acquire. It is the highest quality spectral CT dataset currently available, with most noise having been filtered out by a non-local means (NLM) filter [92]. Examples of slices of the Mouse12 dataset can be found in section 3.1, in Fig. 3.2 and 3.3.

Fig. 5.5 shows that PSNR for this dataset is within acceptable limits and shows that VQ2 can sometimes reach the PSNR of VQ1. This metric, however, does not always reflect the visual quality perceived by the user.

As seen in Fig. 5.6, which shows a close-up comparison between compressed and uncompressed versions of the 23 keV energy bin of the Mouse12 dataset, blocky artifacts are present when VQ2 is used. These artifacts detract from the smoothness of some surfaces but overall do not severely affect image quality. This is due to the fact that this is a large dataset and individual blocks of 4^3 voxels that

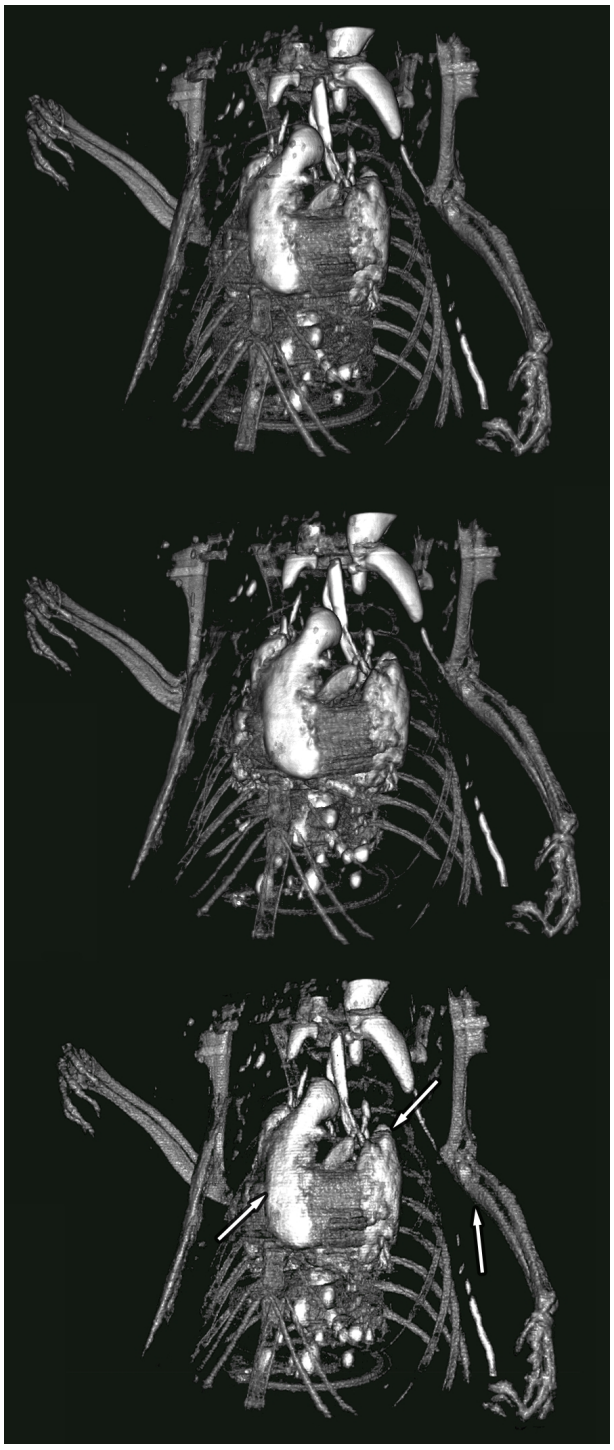

Figure 5.6: Comparison between the compressed and uncompressed versions of the 23 keV energy bin of the Mouse12 dataset rendered with MARSCUDAVR. Top: uncompressed bin. Middle: compressed with VQ1. Bottom: compressed with VQ2, visible distortion highlighted with arrows.

may be poorly compressed (for example, if the boundaries between two blocks are very prominent) do not influence the final image to the same degree as they would influence a smaller dataset. Overall, however, VQ2 works well for an algorithm that compresses data at the ratio of 64:3. The visual quality offered by VQ1 is even better: there is very little visual difference between the compressed and uncompressed energy bins.

5.2.2.2 FatCaFe Dataset

This dataset has been obtained by scanning a *phantom*, which is an object specifically designed to test and calibrate a medical imaging system. In this case, six small cylinders have been filled with sunflower oil, air, water and iron and calcium solutions and scanned with the MARS-CT system. This dataset consists of six energy bins, with each bin containing 436x436x220 voxels. This dataset is small enough to be visualised without compression, but it has been processed with VQ1 and VQ2 and visualised with MARSCUDAVR due to the lack of large spectral CT datasets available for study.

PSNR of the FatCaFe dataset compressed with both algorithms is shown in Fig. 5.7 and rendered images of energy bin 2 of this dataset are shown in Fig. 5.8. These images are only presented as illustrations, as FatCaFe is so noisy that visualisation is unlikely to yield any meaningful results. Currently, the study of this dataset is restricted to mathematical and statistical methods.

In order to clearly illustrate the degree to which noise affects the visualisation

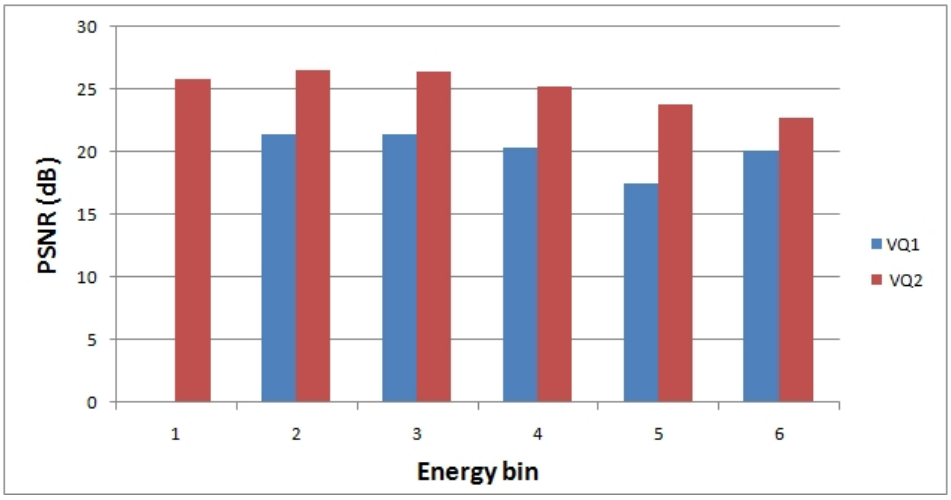

Figure 5.7: PSNR of six bins of the FatCaFe dataset compressed with VQ1 and VQ2.

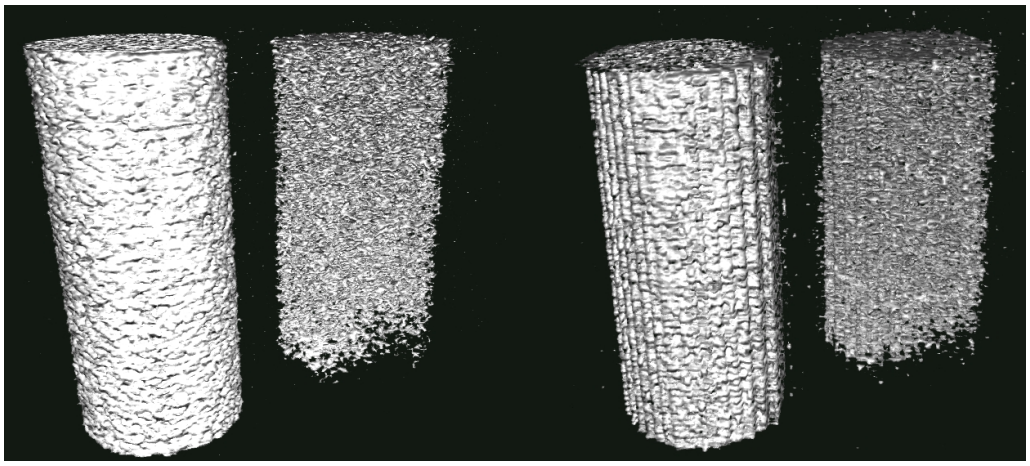

Figure 5.8: Energy bin 2 of the FatCafe dataset compressed with VQ1 (left) and VQ2 (right) and visualised with MARSCUDAVR.

of spectral CT data, FatCaFe can be compared to the best spectral CT dataset currently available: Mouse12. As shown in Fig. 5.9, the slices comprising the Mouse12 dataset are of much higher quality and contain less noise that disrupts compression and visualisation. FatCaFe, on the other hand, contains ring artifacts [93] and a large amount of Poisson noise [94], which is a problem for all photon-counting detectors, including the Medipix series.

In summary, the effectiveness of both VQ and VQ2 is limited due to the large amounts of noise present in the FatCaFe dataset which, in its current state, is too

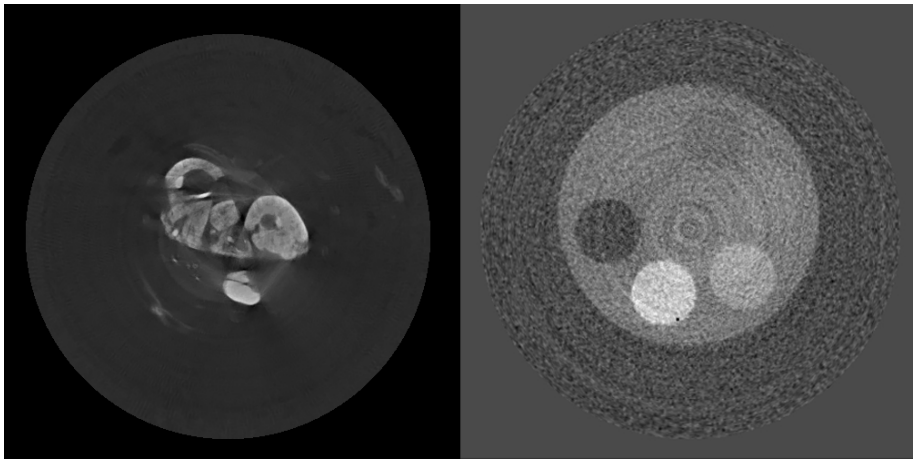

Figure 5.9: Comparison of the Mouse12 and FatCaFe datasets. Left: a representative slice from the Mouse12 dataset. Right: a representative slice from the FatCaFe dataset.

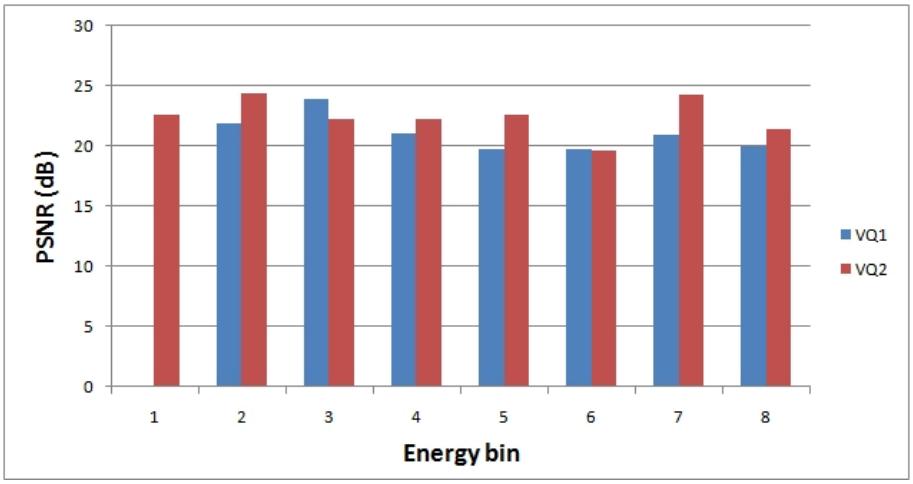

Figure 5.10: PSNR of eight bins of the Phantom dataset compressed with VQ1 and VQ2.

noisy to be visualised. FatCaFe, however, is useful for illustrating the problem of compressing and visualising low quality datasets and shows that further research into image processing techniques is required to improve the quality of raw slice data.

5.2.2.3 Phantom Dataset

This dataset is another phantom created to test the MARS-CT scanner. The aim of this study was to separate five gold nano-particle solutions of varying concentrations [19]. It is a small dataset (280x280x200 voxels) and requires no compression, even on older hardware. However, it comprises 23 energy bins, of which eight were

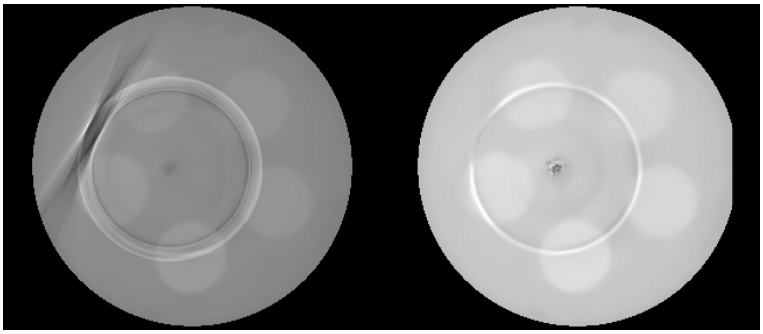

Figure 5.11: Artifacts present in the slices of the Phantom dataset. Left: ring artifacts and reconstruction artifacts (top left) in slice 197 of energy bin 4. Right: ring artifacts in slice 122 of energy bin 8.

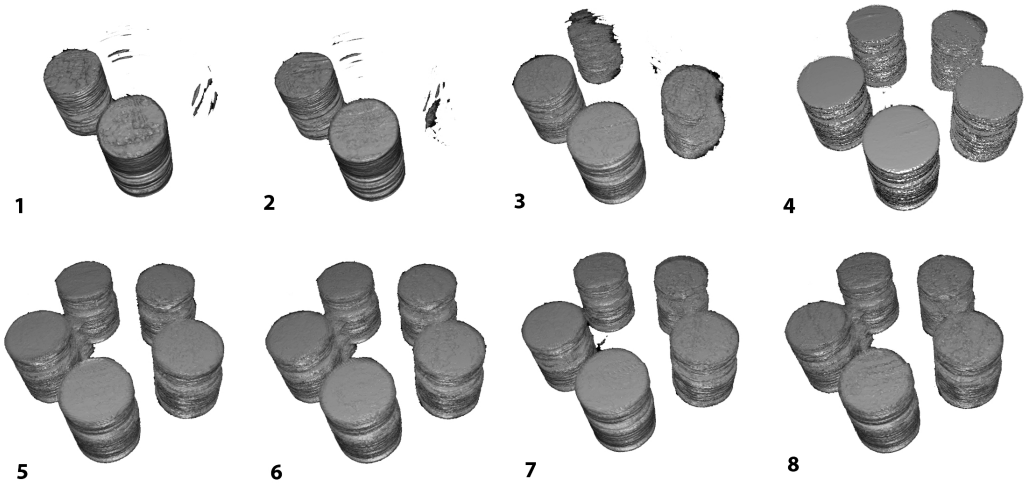

Figure 5.12: Eight energy bins of the Phantom dataset compressed with VQ1 and visualised separately with MARSCUDAVR.

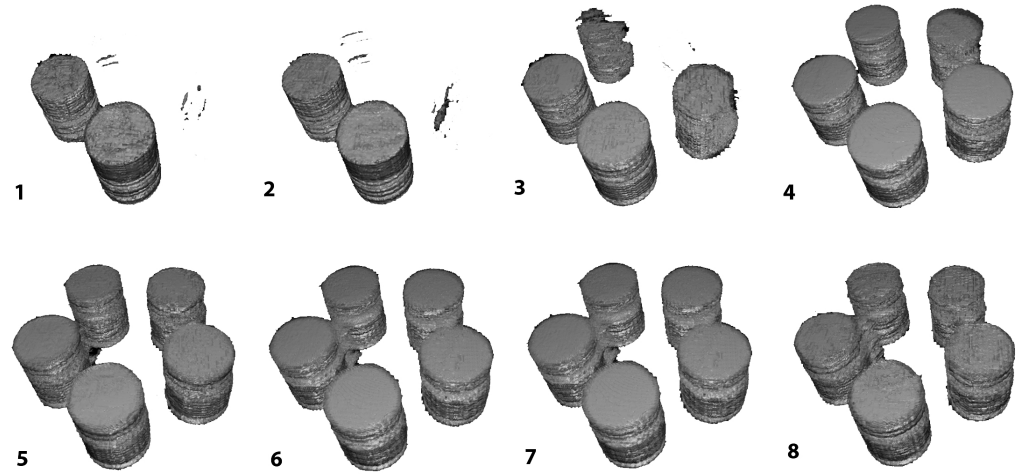

Figure 5.13: Eight energy bins of the Phantom dataset compressed with VQ2 and visualised separately with MARSCUDAVR.

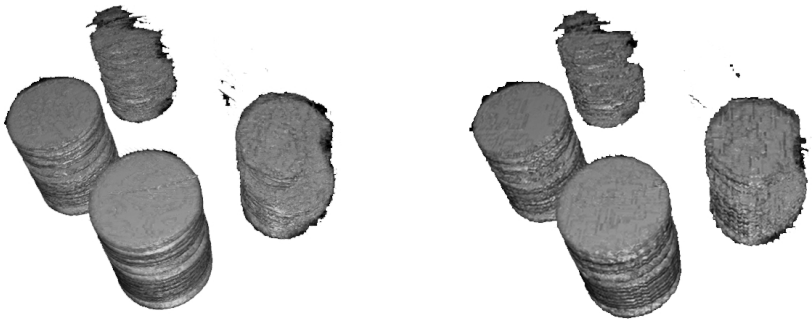

Figure 5.14: Left: bin 3 of the Phantom dataset compressed with VQ1 and visualised with MARSCUDAVR. Right: same bin compressed with VQ2.

chosen for visualisation and processed with VQ1 and VQ2 before being visualised (examples of raw slices of the eight energy bins are shown in Fig. 3.1). This dataset, while not being as noisy as FatCaFe, still suffers from ring and reconstruction artifacts, as seen in Fig. 5.11. Rendered images of all energy bins are shown in Fig. 5.12 and 5.13.

PSNR measured for the Phantom dataset compressed with both algorithms is shown in Fig. 5.10. As expected, the quality of compression with VQ1 decreases as more energy bins are added to the dataset. This is clearly because it is harder to generate a codebook that describes a set of longer vectors than a codebook for the same number of shorter vectors. As vector length increases, so does the complexity and more distortion is introduced. For standard 4-8 energy bin spectral CT datasets, however, generating codebooks and quantising difference vectors is acceptable, provided that datasets are not extremely noisy, as noise disrupts the patterns that may be present in data and reduces the correlation between energy bins.

As seen in Fig. 5.14, noticeable block artifacts appear in small volume datasets compressed by VQ2. The reason, as explained above, is because VQ2 is based on compressing individual blocks of 4^3 voxels and in small datasets each block is relatively large compared to the overall size. Therefore, boundaries between blocks become much easier to notice, causing block artifacts and visible image distortion. Large datasets, such as Mouse12, do not suffer from this problem to the same degree because each block is smaller relative to the size of the full dataset.

5.2.2.4 Conclusion

VQ1 provides substantially better image quality while also being more computationally efficient than VQ2. The reason for better quality is that VQ1 considers the correlation between multiple energy bins in a spectral CT dataset, while VQ2 only finds common features in small blocks (4^3) of voxels, without attempting to utilise the features of the entire dataset to its advantage. However, both algorithms are adapted for rapid decompression on GPUs. This requirement limits the range of techniques that can be applied and excludes the possibility of using lossless and high-quality lossy compression methods.

Overall, VQ1 can be considered the universal algorithm for spectral CT data compression which possesses a significant advantage in terms of image quality and decompression speed. VQ2, on the other hand, can be utilised for quick, lower-quality visualisation on hardware with small amounts of graphics memory. These conclusions correspond to the original design goals: VQ1 has been created with versatility and quality in mind and VQ2 has been designed as an alternative to use

in cases where quality may need to be sacrificed for a higher compression ratio.

Another valuable conclusion that can be drawn from this evaluation is that noise in reconstructed spectral CT datasets severely degrades the performance of both compression schemes and that VQ1 suffers from more distortion than VQ2 if a noisy dataset is used. VQ1 works best if there is a large number of repeating vectors in the difference volume, which, due to the nature of spectral CT (section 3.1), can be expected in high-quality datasets such as Mouse12.

Noisy datasets such as FatCaFe and Phantom, on the other hand, have fewer common sequences if difference volumes are created and analysed, making it substantially harder to generate good codebooks. The MARS project, however, is making rapid advances in spectral CT technology and it can be expected that the quality of datasets acquired by the MARS-CT scanner will improve significantly in the next few years.

5.3 Comparison With MARSCTExplorer

This section compares the performance and visual quality of MARSCUDAVR to that of MARSCTExplorer. While some performance metrics have already been gathered and presented, the aim of this section is to discuss the reasons behind them and investigate some of the factors influencing the frame rate of each application.

MARSCTExplorer is the software currently used to visualise spectral CT datasets generated by the MARS-CT scanner [19]. It renders directly from uncompressed 3D textures stored within the memory of a single GPU and allows the user to combine energy bins using arithmetic and logical operators. It is difficult to directly compare MARSCUDAVR and MARSCTExplorer because the two applications are different in terms of their design, functionality and internal architecture. For example, different algorithms are used for gradient estimation and classification by the transfer function. Therefore, only an approximate comparison can be made.

To ensure a fair comparison, all tests have been performed on raw volume data with no transfer functions or combinations being applied. MARSCTExplorer runs at 25 fps while visualising the Mouse12 dataset [19], while MARSCUDAVR is 2-10 times slower, depending on the algorithm used for spectral CT data compression and on quality settings (see, for example, section 4.4.3 and table 4.3).

In terms of performance, MARSCTExplorer is vastly superior to MARSCUDAVR, but such difference must be expected: as explained in section 4.3.2, MARSCUDAVR must perform trilinear interpolation in software, whereas MARSCTExplorer can use built-in GPU functionality. MARSCUDAVR must also decompress

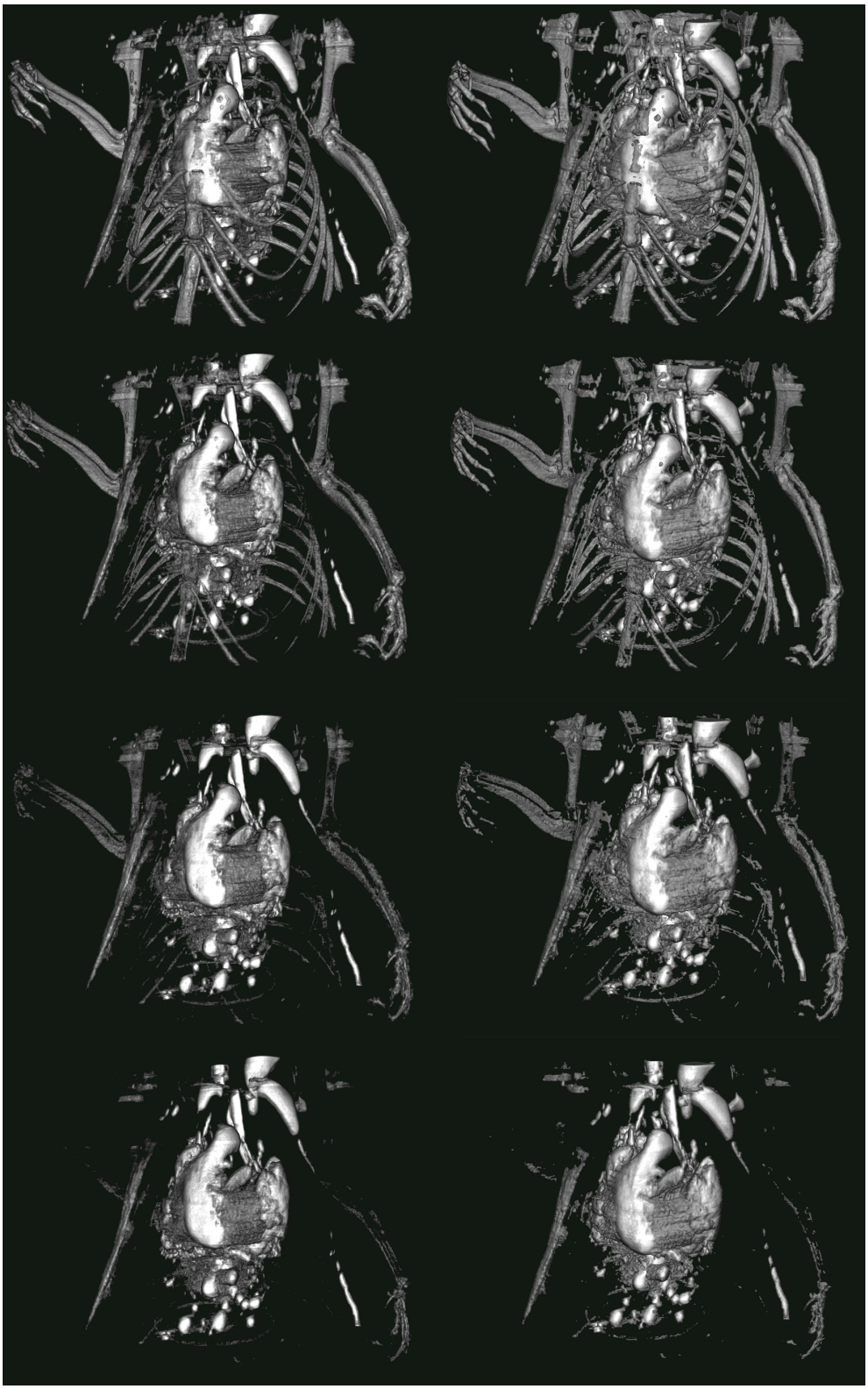

Figure 5.15: Comparison of the Mouse12 dataset compressed with VQ1 and rendered with MARSCUDAVR (left) and MARSCTExplorer (right). From top to bottom: 15 keV, 23 keV, 30 keV and 35 keV energy bins. MARSCTExplorer images courtesy of Niels de Ruiter.

each voxel before use which leads to a significant performance decrease.

As shown in Fig. 5.15, the visual quality of the two applications is comparable when VQ1 is used to compress spectral CT data. MARSCTExplorer is inherently faster, so it can use a slower but higher quality cubic spline-based interpolation scheme. As a result, some surfaces may appear smoother, but in general the difference is minor.

Overall, it has been shown that MARSCUDAVR is capable of interactive visualisation of spectral CT datasets, with visual quality being similar to MARSCTExplorer. The most important cause of differences between images rendered by these applications is compression, which will inevitably introduce some distortion into any image rendered by MARSCUDAVR. However, MARSCUDAVR's primary advantage lies in its ability to visualise, given the same hardware, datasets 4-8 times larger than those supported by MARSCTExplorer.

5.4 Performance on Different CUDA Devices

Device architecture has a significant impact on the speed of a CUDA kernel. While shader, core and memory clock speeds and the number of cores may appear to be the determining factors in kernel performance, in reality the architecture of a GPU also plays an important role.

Volume raycasting has already been identified as a task that does not benefit from parallelisation on a GPU as much as some other problems because of the random way in which sampling rays access device memory. Algorithms which access memory in a highly structured and predictable way are much more suitable for GPU acceleration. In a recent study by Lee at al [51], optimised CPU and GPU implementations of the volume raycasting algorithm were compared, with the GPU version only being 1.6x faster. However, Lee et al. performed the comparison using GT200 series NVIDIA GPUs (Compute Capability 1.3), which are not as well-adapted for GPGPU computation as the newer Fermi-series cards.

Table 5.2: Performance of MARSCUDAVR on different GPUs. Average frame rates measured for the Mouse12 dataset at high quality settings.

Architecture	**VQ1**	**VQ2**
Quadro FX5800 (Compute Capability 1.3)	0.71 fps	0.34 fps
GeForce GTX470 (Compute Capability 2.0)	3.74 fps	1.58 fps
Performance improvement (%)	426%	365%

The most modern CUDA devices (Fermi GPUs that support CUDA Compute Capability 2.0 and 2.1) are much better suited for GPGPU work due to multiple levels of caching and highly optimised accesses to global memory [35], making them especially suitable for algorithms such as volume raycasting that may access any location in global memory without ever utilising a regular access pattern.

Results shown in table 5.2 confirm this theory. An older Quadro FX5800 GPU was compared to a newer GeForce GTX470 (see section 4.1 for detailed specifications). The Quadro FX5800 has 240 CUDA cores compared to the GTX470's 448 cores and both cards have very similar clock speeds. The performance difference, however, is dramatic: MARSCUDAVR using VQ1 or VQ2 is 300-400% faster on the GTX470 than on the Quadro FX5800.

Newer cards are far more suitable for GPGPU computation due to the differences described above. This is the reason why this thesis has emphasised the need to develop algorithms specifically tailored to the latest GPU hardware. New GPUs are faster in general and the architectural differences from older GPUs may also invalidate some established programming and optimisation techniques and favour different ones, or even lead to the creation of novel methods.

5.5 Summary

This chapter has presented a brief evaluation of spectral CT data compression using the VQ1 and VQ2 algorithms and its subsequent visualisation with MARSCUDAVR. In particular:

- The effect of codebook size on the quality of compression with VQ1 and VQ2 has been noted.

- Three compressed spectral CT datasets have been visualised and studied. Conclusions about the proper way of applying compression have been drawn. The necessary attributes that a spectral CT dataset must possess in order to be compressed at a reasonable quality level have been described.

- The effects of noise on compression and visualisation of spectral CT data have been noted.

- A comparison of visual quality and frame rates of MARSCUDAVR and MARSCTExplorer has been made and the reasons for any differences between the two applications have been explained.

- The effect of GPU architecture on the frame rate of a CUDA-based volume rendering application has been studied.

It must also be noted that this evaluation was hampered by the lack of high quality spectral CT datasets on which compression algorithms could be tested. Only a single dataset, Mouse12, is relatively noise-free and large enough to enable detailed study of compression. However, more high quality multi-gigabyte datasets suitable for compression with VQ1 and VQ2 will undoubtedly be acquired in the future by the MARS team. Continued evaluation of both algorithms and MARSCUDAVR, therefore, will remain a priority in the near future.

Chapter VI

Conclusion

The work described in this thesis confirms the possibility of rendering large spectral CT datasets on consumer-grade GPU hardware by integrating compression (as part of pre-processing) and real-time in-core decompression into a volume rendering application.

Two algorithms for doing so have been presented in this thesis and evaluated against each other and against existing OpenGL-based rendering software. These algorithms are sufficiently generic that they can be applied to any type of data that is reasonably similar to spectral CT datasets. In addition, methods for improving the speed of CUDA-accelerated visualisation algorithms have been presented and CUDA-specific optimisation of visualisation code has been discussed.

Over the course of this research the following contributions have been achieved:

- Characteristics of spectral CT datasets have been examined in detail and properties enabling compression have been identified. In particular, the correlation between data from different energy bins has been noted as being a potential basis for various compression methods.

- VQ1, a compression scheme specifically designed for spectral CT datasets, has been implemented and tested. Evaluation shows that VQ1 provides reasonable image quality and enables interactive visualisation of multi-gigabyte spectral CT datasets on consumer-grade hardware.

- VQ2, an algorithm originally designed for 3D RGB texture compression, has been adapted and optimised for spectral CT data compression. Results indicate that it can be used as an alternative to VQ1 when rendering large datasets on GPUs with lower amounts of video memory.

- MARSCUDAVR, a CUDA-accelerated volume raycasting application that renders directly from uncompressed volume data or from spectral CT datasets compressed with VQ1 or VQ2 has been developed over the course of this project. Its implementation details have been discussed and it has been

demonstrated that MARSCUDAVR is able to achieve interactive frame rates and similar visual quality to standard spectral CT visualisation software.

- Visualisation of compressed and uncompressed spectral CT datasets using CUDA has been investigated and its advantages and disadvantages have been identified.

- Performance issues caused by rendering from compressed spectral CT data formats have been explained. The frame rate of MARSCUDAVR has been significantly improved through the use of spatial data structures and GPU-specific optimisation techniques.

In conclusion, this research shows that, by utilising appropriate compression and acceleration techniques, interactive visualisation of multi-gigabyte spectral CT datasets can be achieved on most high performance yet relatively inexpensive and widely available consumer-grade GPU hardware.

6.1 Future Work

This project has explored an entirely new topic by considering real-time visualisation of compressed spectral CT datasets, but this work must be thought of as being only the start, as numerous avenues for further research remain. More efficient and higher-quality compression schemes can be created and improvements to the speed of visualisation algorithms appear to be possible. In addition, several other research directions can be thought of as being logical extensions of the work described in this thesis.

First of all, regardless of the methods used for compressing and visualising spectral CT datasets, compression for the purposes of storage on permanent media (such as hard drives) can be investigated. Promising methods such as wavelet-based or 4D zerotree compression could be highly efficient in reducing storage requirements for spectral CT datasets at the price of negligible reduction in image quality, or even no reduction at all.

Storing data in a compressed format on the hard drive may become useful when the MARS-CT scanner is developed further and the size of its datasets increases. As demonstrated by this thesis, there is a high degree of redundancy in spectral CT datasets and highly specialised compression methods that are not constrained by the need for real-time decompression can certainly be developed.

As mentioned in Chapter 5, noise and visual artifacts pose a big problem during the visualisation and study of spectral CT datasets (whether compressed or un-

compressed). Further research into image processing, filtering and reconstruction algorithms for spectral CT data will therefore be highly beneficial.

Segmentation and material decomposition is another relevant research area. Reliably segmenting medical images in general is difficult, but spectral CT data has the advantage of having multiple descriptions of the same object, which can in theory be used to determine the physical properties of materials being studied. If a material in a spectral CT dataset can be identified with a high degree of confidence, then reliable segmentation can be achieved.

The benefits offered by well-designed segmentation algorithms are significant as segmented data can aid medical professionals in making a diagnosis and improve compression by reducing large and complex datasets to a set of pre-defined materials.

Non-real-time photorealistic rendering on high-performance computing (HPC) hardware is another potential research direction. Such approach would focus primarily on improving the visual quality of generated images, disregarding the speed of visualisation. A combination of lower quality real-time visualisation using GPU hardware and subsequent high quality rendering of regions of interest on HPC hardware may combine the best aspects of both techniques and improve the scientific and medical utility of the MARS-CT system and spectral CT imaging in general.

References

[1] J. Butzer, A. Butler, N. Cook, P. Butler, F. Ross, N. Schleich, J. Selkirk, R. Watts, J. Meyer, N. Scott, P. Bones, D. van Leeuwen, S. Hemmingsen, T. Melzer, N. Anderson, and MARS-CT Team, "Mars: a 3d spectroscopic x-ray imaging device based on medipix," 2008.

[2] A. Butler, N. Anderson, R. Tipples, N. Cook, R. Watts, J. Meyer, A. Bell, T. Melzer, and P. Butler, "Bio-medical x-ray imaging with spectroscopic pixel detectors," *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 591, no. 1, pp. 141 – 146, 2008. Radiation Imaging Detectors 2007 - Proceedings of the 9th International Workshop on Radiation Imaging Detectors.

[3] K. Berg, J. Carr, M. Clark, N. Cook, N. Anderson, N. Scott, A. Butler, P. Butler, and A. Butler, "Pilot study to confirm that fat and liver can be distinguished by spectroscopic tissue response on a medipix-all-resolution-system-ct (mars-ct)," in *ADVANCED MATERIALS AND NANOTECHNOLOGY: Proceedings of the International Conference (AMN-4)*, pp. p106–110, American Institute of Physics Conference Proceedings, 2009.

[4] MARS-CT Project, "MARS-CT Project." `http://wiki.canterbury.ac.nz/display/MARSCT/Home`, 2011.

[5] J. Bushberg, *The essential physics of medical imaging.* Lippincott Williams & Wilkins, 2002.

[6] G. Hounsfield, "Method and apparatus for measuring x- or -radiation absorption or transmission at plural angles and analyzing the data," December 1973.

[7] R. H. T. Bates and T. M. Peters, "Towards improvements in tomography," *N. Z. J. Sci*, vol. 14, pp. 883–896, 1971.

[8] W. A. Kalender, "X-ray computed tomography," *Physics in Medicine and Biology*, vol. 51, no. 13, p. R29, 2006.

[9] J. Giersch and J. Durst, "Monte carlo simulations in x-ray imaging," *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 591, no. 1, pp. 300 – 305, 2008.

[10] N. Anderson, A. Butler, N. Scott, N. Cook, J. Butzer, N. Schleich, M. Firsching, R. Grasset, N. de Ruiter, M. Campbell, and P. Butler, "Spectroscopic (multi-energy) ct distinguishes iodine and barium contrast material in mice," *European Radiology*, vol. 20, pp. 2126–2134, 2010. 10.1007/s00330-010-1768-9.

[11] L. G. Brown, "A survey of image registration techniques," *ACM Comput. Surv.*, vol. 24, pp. 325–376, Dec. 1992.

[12] R. Zainon, A. P. H. Butler, N. J. Cook, J. S. Butzer, N. Schleich, N. D. Ruiter, L. Tlustos, M. J. Clark, R.Heinz, and P. H. Butler, "Construction and operation of the mars-ct scanner," *Internetworking Indonesia Journal*, vol. 2, pp. 2–10, 2010.

[13] M. F. Walsh, A. M. T. Opie, J. P. Ronaldson, R. M. N. Doesburg, S. J. Nik, J. L. Mohr, R. Ballabriga, A. P. H. Butler, and P. H. Butler, "First ct using medipix3 and the mars-ct-3 spectral scanner," *Journal of Instrumentation*, vol. 6, no. 01, p. C01095, 2011.

[14] R. Ballabriga, M. Campbell, E. H. M. Heijne, X. Llopart, and L. Tlustos, "The Medipix3 Prototype, a Pixel Readout Chip Working in Single Photon Counting Mode With Improved Spectrometric Performance," *IEEE Transactions on Nuclear Science*, vol. 54, pp. 1824–1829, Oct. 2007.

[15] Merge Healthcare. `http://www.merge.com`, 2011.

[16] M. Levoy, "Display of surfaces from volume data," *Computer Graphics and Applications, IEEE*, vol. 8, pp. 29 –37, may 1988.

[17] P. Lacroute and M. Levoy, "Fast volume rendering using a shear-warp factorization of the viewing transformation," in *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, SIGGRAPH '94, (New York, NY, USA), pp. 451–458, ACM, 1994.

[18] L. Westover, "Footprint evaluation for volume rendering," *SIGGRAPH Comput. Graph.*, vol. 24, pp. 367–376, September 1990.

[19] N. de Ruiter, "Gpu accelerated intermixing as a framework for interactively visualizing spectral ct data," Master's thesis, University of Canterbury, 2011.

[20] W. E. Lorensen and H. E. Cline, "Marching cubes: A high resolution 3d surface construction algorithm," *COMPUTER GRAPHICS*, vol. 21, no. 4, pp. 163–169, 1987.

[21] T. Möller, K. Mueller, Y. Kurzion, R. Machiraju, and R. Yagel, "Design of accurate and smooth filters for function and derivative reconstruction," in *Proceedings of the 1998 IEEE symposium on Volume visualization*, VVS '98, (New York, NY, USA), pp. 143–151, ACM, 1998.

[22] S. Marschner and R. Lobb, "An evaluation of reconstruction filters for volume rendering," in *Visualization, 1994., Visualization '94, Proceedings., IEEE Conference on*, pp. 100 –107, CP10, oct 1994.

[23] R. Fernando, *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*, ch. Chapter 39. Volume Rendering Techniques. Addison-Wesley, 2004.

[24] J. Kruger and R. Westermann, "Acceleration techniques for gpu-based volume rendering," in *Proc. IEEE Visualization VIS 2003*, pp. 287–292, 2003.

[25] X. Tang and W. Pearlman, "Three-dimensional wavelet-based compression of hyperspectral images," in *Hyperspectral Data Compression* (G. Motta, F. Rizzo, and J. A. Storer, eds.), pp. 273–308, Springer US, 2006.

[26] L. Zeng, C. P. Jansen, S. Marsch, M. Unser, and P. R. Hunziker, "Four-dimensional wavelet compression of arbitrarily sized echocardiographic data," *IEEE Transactions on Medical Imaging*, vol. 21, no. 9, pp. 1179–1187, 2002.

[27] W. Cai and G. Sakas, "Data intermixing and multi-volume rendering," *Comput. Graph. Forum*, pp. 359–368, 1999.

[28] M. Andrecut, "Parallel gpu implementation of iterative pca algorithms," *Journal of Computational Biology*, vol. 16, no. 11, p. 45, 2008.

[29] A. Butler, N. Cook, N. Schleich, J. Butzer, P. Bones, P. Butler, and MARS-CT Team, "Processing of spectral x-ray data using principal components analysis," *Nuclear Instruments and Methods in Physics Research Section A*, vol. 633, pp. S140–S142, 2010.

[30] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krger, A. Lefohn, and T. J. Purcell, "A survey of general-purpose computation on graphics hardware," *Computer Graphics Forum*, vol. 26, no. 1, pp. 80–113, 2007.

[31] N. L., H. M., and P. J., *GPU Gems 3*, ch. Fast N-Body Simulation with CUDA, pp. 677–695. Addison-Wesley, 2007.

[32] L. Marsalek, A. Hauber, and P. Slusallek, "High-speed volume ray casting with cuda," in *IEEE Symposium on Interactive Ray Tracing, 2008. RT 2008.*, p. 185, aug. 2008.

[33] B. Kainz, M. Grabner, A. Bornik, S. Hauswiesner, J. Muehl, and D. Schmalstieg, "Ray casting of multiple volumetric datasets with polyhedral boundaries on manycore gpus," in *ACM SIGGRAPH Asia 2009 papers*, SIGGRAPH Asia '09, (New York, NY, USA), pp. 152:1–152:9, ACM, 2009.

[34] Y.-L. Huang, Y.-C. Shen, and J.-L. Wu, "Scalable computation for spatially scalable video coding using nvidia cuda and multi-core cpu," in *Proceedings of the 17th ACM international conference on Multimedia*, MM '09, (New York, NY, USA), pp. 361–370, ACM, 2009.

[35] NVIDIA Corporation, "Cuda programming guide (version 4.0)." `http://developer.download.nvidia.com/compute/cuda/4_0_rc2/toolkit/docs/CUDA_C_Programming_Guide.pdf`, March 2011.

[36] Benchmark Reviews, "Nvidia fermi architecture." `http://benchmarkreviews.com/index.php?option=com_content&task=view&id=440&Itemid=63&limit=1&limitstart=3`, January 2010.

[37] NVIDIA Corporation, "Nvidia's next generation compute architecture: Fermi." `http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIAFermiComputeArchitectureWhitepaper.pdf`, 2008.

[38] M. Weiler, M. Kraus, M. Merz, and T. Ertl, "Hardware-based ray casting for tetrahedral meshes," in *Proc. IEEE Visualization VIS 2003*, pp. 333–340, 2003.

[39] L. Weiguo, B. Schmidt, G. Voss, and W. Muller-Wittig, "Streaming algorithms for biological sequence alignment on gpus," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 18, pp. 1270 –1281, sept. 2007.

[40] K. Perumalla, "Discrete-event execution alternatives on general purpose graphical processing units (gpgpus)," in *Principles of Advanced and Distributed Simulation, 2006. PADS 2006. 20th Workshop on*, pp. 74 –81, 2006.

[41] NVIDIA Corporation, "Ptx: Parallel thread execution isa version 2.0." `http://developer.download.nvidia.com/compute/cuda/3_0/toolkit/docs/ptx_isa_2.0.pdf`, January 2010.

[42] NVIDIA Corporation, "Cuda c best practices guide." `http://developer.download.nvidia.com/compute/cuda/4_0_rc2/toolkit/docs/CUDA_C_Best_Practices_Guide.pdf`, March 2011.

[43] B. Jang, D. Kaeli, S. Do, and H. Pien, "Multi gpu implementation of iterative tomographic reconstruction algorithms," in *Biomedical Imaging: From Nano to Macro, 2009. ISBI '09. IEEE International Symposium on*, pp. 185 –188, 28 2009-july 1 2009.

[44] V. Volkov, "Better performance at lower occupancy," in *GPU Technology Conference 2010*, 2010.

[45] V. Volkov and J. W. Demmel, "Benchmarking gpus to tune dense linear algebra," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, (Piscataway, NJ, USA), pp. 31:1–31:11, IEEE Press, 2008.

[46] NVIDIA Corporation, "Cuda toolkit 4.0." `http://developer.nvidia.com/cuda-toolkit-40`, 2011.

[47] M. Smelyanskiy, D. Holmes, J. Chhugani, A. Larson, D. M. Carmean, D. Hanson, P. Dubey, K. Augustine, D. Kim, A. Kyker, V. W. Lee, A. D. Nguyen, L. Seiler, and R. Robb, "Mapping high-fidelity volume rendering for medical imaging to cpu, gpu and many-core architectures," *IEEE Transactions on Visualization and Computer Graphics*, vol. 15, pp. 1563–1570, Nov. 2009.

[48] Khronos Group, "Opencl - the open standard for parallel programming of heteregeneous systems." `http://www.khronos.org/opencl/`, 2011.

[49] N. K. Govindaraju, B. Lloyd, Y. Dotsenko, B. Smith, and J. Manferdelli, "High performance discrete fourier transforms on graphics processors," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, (Piscataway, NJ, USA), pp. 2:1–2:12, IEEE Press, 2008.

[50] M. Silberstein, A. Schuster, D. Geiger, A. Patney, and J. D. Owens, "Efficient computation of sum-products on gpus through software-managed cache," in *Proceedings of the 22nd annual international conference on Supercomputing*, ICS '08, (New York, NY, USA), pp. 309–318, ACM, 2008.

[51] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey, "Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu," *SIGARCH Comput. Archit. News*, vol. 38, pp. 451–460, June 2010.

[52] C. Bajaj, I. Ihm, and S. Park, "3d rgb image compression for interactive applications," *ACM Trans. Graph.*, vol. 20, pp. 10–38, January 2001.

[53] E. Delp and O. Mitchell, "Image compression using block truncation coding," *IEEE Transactions on Communications*, vol. 27, no. 9, pp. 1335–1342, 1979.

[54] D.-M. Liou, Y. Huang, and N. Reynolds, "A new microcomputer based imaging system with c3 technique," in *TENCON 90. 1990 IEEE Region 10 Conference on Computer and Communication Systems*, pp. 555 –559 vol.2, sep 1990.

[55] S3 Inc, "System and method for fixed-rate block-based image compression with inferred pixel values," September 1999.

[56] Khronos Group, "GL EXT texture compression s3tc." `http://www.opengl.org/registry/specs/EXT/texture_compression_s3tc.txt`, 2009.

[57] Y. Linde, A. Buzo, and R. Gray, "An algorithm for vector quantizer design," *Communications, IEEE Transactions on*, vol. 28, pp. 84 – 95, jan 1980.

[58] A. C. Beers, M. Agrawala, and N. Chaddha, "Rendering from compressed textures," in *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, SIGGRAPH '96, (New York, NY, USA), pp. 373–378, ACM, 1996.

[59] J. Schneider and R. Westermann, "Compression domain volume rendering," in *VIS '03: Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, (Washington, DC, USA), p. 39, IEEE Computer Society, 2003.

[60] S. Mallat, "A theory for multiresolution signal decomposition: the wavelet representation," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 11, pp. 674 –693, jul 1989.

[61] D. Montgomery, F. Murtagh, and A. Amira, "A wavelet based 3d image compression system," in *Signal Processing and Its Applications, 2003. Proceedings. Seventh International Symposium on*, vol. 1, pp. 65 – 68 vol.1, july 2003.

[62] J. Shapiro, "Smart compression using the embedded zerotree wavelet (ezw) algorithm," in *Signals, Systems and Computers, 1993. 1993 Conference Record of The Twenty-Seventh Asilomar Conference on*, pp. 486 –490 vol.1, nov 1993.

[63] B. J. Erickson, "Irreversible compression of medical images," *Journal of Digital Imaging*, vol. 15, pp. 5–14, 2002. 10.1007/s10278-002-0001-z.

[64] N. Thomos, N. V. Boulgouris, and M. G. Strintzis, "Optimized transmission of jpeg2000 streams over wireless channels," *IEEE Transactions on Image Processing*, vol. 15, no. 1, pp. 54–67, 2006.

[65] A. Eskicioglu and P. Fisher, "Image quality measures and their performance," *Communications, IEEE Transactions on*, vol. 43, pp. 2959 –2965, dec 1995.

[66] P. C. Cosman, R. M. Gray, and R. A. Olshen, "Evaluating quality of compressed medical images: Snr, subjective rating, and diagnostic accuracy," *Proceedings of the IEEE*, vol. 82, no. 6, pp. 919–932, 1994.

[67] J. P. Fritsch, R. Brennecke, M. Lang, U. Renneisen, M. Haude, L. Koch, R. Erbel, and J. Meyer, "New techniques for visualization of losses due to image compression in grayscale medical still images," in *Proc. Computers in Cardiology 1992*, pp. 271–273, 1992.

[68] V. Savcenko, B. Erickson, K. Persons, N. Campeau, J. Huston, C. Wood, and S. Schreiner, "An evaluation of jpeg and jpeg 2000 irreversible compression algorithms applied to neurologic computed tomography and magnetic resonance images," *Journal of Digital Imaging*, vol. 13, pp. 183–185, 2000. 10.1007/BF03167656.

[69] T. H. Karson, S. Chandra, A. Morehead, S. E. Nissen, and J. D. Thomas, "Digital compression of echocardiographic images: is it viable?," in *Proc. Computers in Cardiology 1993*, pp. 831–834, 1993.

[70] S. E. Ghrare, M. A. M. Ali, M. Ismail, and K. Jumari, "Diagnostic quality of compressed medical images: Objective and subjective evaluation," in *Proc. Second Asia Int. Conf. Modeling & Simulation AICMS 08*, pp. 923–927, 2008.

[71] P. Ning and L. Hesselink, "Fast volume rendering of compressed data," in *Visualization, 1993. Visualization '93, Proceedings., IEEE Conference on*, pp. 11 –18, oct 1993.

[72] A. Gersho and A. M. Gray, *Vector Quantization and Signal Processing*. Kluwer Academic Publishers, 1991.

[73] M. Hadwiger, J. M. Kniss, C. Rezk-salama, D. Weiskopf, and K. Engel, *Real-time Volume Graphics*. Natick, MA, USA: A. K. Peters, Ltd., 2006.

[74] R. Westermann and B. Sevenich, "Accelerated volume ray-casting using texture mapping," in *Proc. Visualization VIS '01*, pp. 271–278, 2001.

[75] D. Ruijters, B. M. ter Haar-Romeny, and P. Suetens, "Accuracy of gpu-based b-spline evaluation," in *Proceedings of the Tenth IASTED International Conference on Computer Graphics and Imaging*, CGIM '08, (Anaheim, CA, USA), pp. 117–122, ACTA Press, 2008.

[76] R. Fernando and M. Kilgard, *The CG Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Addison-Wesley, 2003.

[77] J. Durkin and J. Hughes, "Nonpolygonal isosurface rendering for large volume datasets," in *Visualization, 1994., Visualization '94, Proceedings., IEEE Conference on*, pp. 293 –300, CP33, oct 1994.

[78] J. F. Blinn, "Models of light reflection for computer synthesized pictures," *SIGGRAPH Comput. Graph.*, vol. 11, pp. 192–198, July 1977.

[79] C. Crassin, F. Neyret, S. Lefebvre, and E. Eisemann, "Gigavoxels : Ray-guided streaming for efficient and detailed voxel rendering," in *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D)*, (Boston, MA, Etats-Unis), ACM, ACM Press, feb 2009. to appear.

[80] D. Laur and P. Hanrahan, "Hierarchical splatting: a progressive refinement algorithm for volume rendering.," in *SIGGRAPH'91*, pp. 285–288, 1991.

[81] S. N. Srihari, "Representation of three-dimensional digital images," *ACM Comput. Surv.*, vol. 13, pp. 399–424, December 1981.

[82] D. R. Horn, J. Sugerman, M. Houston, and P. Hanrahan, "Interactive k-d tree gpu raytracing," in *Proceedings of the 2007 symposium on Interactive 3D graphics and games*, I3D '07, (New York, NY, USA), pp. 167–174, ACM, 2007.

[83] S. Lefebre, S. Hornus, and F. Neyret, *GPU Gems 2*, ch. Octree Textures on the GPU, pp. 595–613. Addison-Wesley, 2005.

[84] M. Tamminen and H. Samet, "Efficient octree conversion by connectivity labeling," *SIGGRAPH Comput. Graph.*, vol. 18, pp. 43–51, January 1984.

[85] W. Li, K. Mueller, and A. Kaufman, "Empty space skipping and occlusion clipping for texture-based volume rendering," in *Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, VIS '03, (Washington, DC, USA), pp. 42–, IEEE Computer Society, 2003.

[86] M. Kraus, M. Strengert, T. Klein, and T. Ertl, "Adaptive sampling in three dimensions for volume rendering on gpus," in *Visualization, 2007. APVIS '07. 2007 6th International Asia-Pacific Symposium on*, pp. 113 –120, feb. 2007.

[87] S. Bergner, T. Moller, D. Weiskopf, and D. J. Muraki, "A spectral analysis of function composition and its implications for sampling in direct volume visualization," *IEEE Transactions on Visualization and Computer Graphics*, vol. 12, pp. 1353–1360, Sept. 2006.

[88] T. Klein, M. Strengert, S. Stegmaier, and T. Ertl, "Exploiting frame-to-frame coherence for accelerating high-quality volume raycasting on graphics hardware," in *Visualization, 2005. VIS 05. IEEE*, pp. 223 – 230, oct. 2005.

[89] R. Yagel and Z. Shi, "Accelerating volume animation by space-leaping," in *Visualization, 1993. Visualization '93, Proceedings., IEEE Conference on*, pp. 62 –69, oct 1993.

[90] NVIDIA Corporation, "SLI - FAQ - GeForce." `http://www.geforce.com/Hardware/Technologies/sli/faq`, 2011.

[91] M.-Y. Chan, Y. Wu, and H. Qu, "Quality enhancement of direct volume rendered images," in *IEEE/EG International Symposium on Volume Graphics (VG'07)*, 2007.

[92] N. Pan, H. Liu, N. de Ruiter, and R. Grasset, "Improving the image quality of spectral ct volume rendering," in *Image and Vision Computing New Zealand, 2009. IVCNZ '09. 24th International Conference*, pp. 203 –208, nov. 2009.

[93] Y.-W. Chen, G. Duan, A. Fujita, K. Hirooka, and Y. Ueno, "Ring artifacts reduction in cone-beam ct images based on independent component analysis," in *Instrumentation and Measurement Technology Conference, 2009. I2MTC '09. IEEE*, pp. 1734 –1737, may 2009.

[94] I. Rodrigues, J. Sanches, and J. Bioucas-Dias, "Denoising of medical images corrupted by poisson noise," in *Image Processing, 2008. ICIP 2008. 15th IEEE International Conference on*, pp. 1756 –1759, oct. 2008.

Appendix A

Appendix A: Difference Images

Figure A.1: Difference images for slice 192 of the Mouse12 dataset, same as in Fig. 3.5, but without adjustments to enhance contrast. Differences between the 15 keV energy bin and, clockwise from top left: the 23, 30 and 35 keV energy bins.

Appendix B

Appendix B: Gradient Estimation Algorithms

```
/* Interpolation function that looks at the given decoded raw volume and
interpolates according to the coordinate provided by the coord parameter
(within a voxel block of size 2x2x2). Also finds the normal of a
sample at the given point. */
__forceinline__ __device__
float tli (float * volume, float3 coord, float3 * normal)
{
    /* This works on the assumption that a block of 2x2x2 voxels is
    looked at, and the sample and normal vector are calculated
    according to the equation below. */

    volatile float sample, H, G, F, E, D, C, B, A;

    H = volume[C000];
    G = volume[C001] - H;
    F = volume[C010] - H;
    E = volume[C100] - H;
    D = volume[C011] - volume[C001] - F;
    C = volume[C101] - volume[C100] - G;
    B = volume[C110] - volume[C010] - E;
    A = volume[C111] - volume[C011] - B - C - E;

    sample = A * coord.x * coord.y * coord.z +
             B * coord.x * coord.y +
             C * coord.x * coord.z +
             D * coord.y * coord.z +
             E * coord.x +
             F * coord.y +
             G * coord.z +
             H;

    normal->x = A * coord.y * coord.z + B * coord.y + C * coord.z + E;
    normal->y = A * coord.x * coord.z + B * coord.x + D * coord.z + F;
    normal->z = A * coord.x * coord.y + C * coord.x + D * coord.y + G;

    return sample;
}
```

Figure B.1: CUDA code for simultaneously calculating the interpolated sample value and the gradient at a sampling position with no extra accesses to global memory. This algorithm provides a fast, low-quality estimate of the gradient vector.

```
//Central differences for lighting

//Move 1 unit forward along the x-axis
SamplingPosition.x += 1
Take sample
Gradient.x = Sample

//Move 2 units back along the x-axis
SamplingPosition.x -= 2
Take sample
Gradient.x -= Sample

//Move 1 unit forward along the x-axis and 1 unit forward along the y-axis
SamplingPosition.x += 1
SamplingPosition.y += 1
Take sample
Gradient.y = Sample

//Move 2 units back along the y-axis
SamplingPosition.y -= 2
Take sample
Gradient.y -= Sample

//Move 1 unit forward along the y-axis and 1 unit forward along the z-axis
SamplingPosition.y += 1
SamplingPosition.z += 1
Take sample
Gradient.z = Sample

//Move 2 units back along the z-axis
SamplingPosition.z -= 2
Take sample
Gradient.z -= Sample
```

Figure B.2: Pseudocode for finding the gradient at a given sampling position using the central differences algorithm. Sampling six times and subtracting the sample values as shown in this figure generates a reasonably accurate estimate of the gradient at a sampling position.

Appendix A

Appendix C: Glossary of Terms

Asymmetric compression - an approach to data or image compression where one operation (compression or decompression) is more computationally expensive than the other. For the purpose of interactive rendering from compressed data, decompression needs to be less computationally expensive than compression. The compression algorithms described in this thesis, VQ1 and VQ2, are both asymmetric and enable fast decompression of volume data on GPU hardware. See section 2.4 for further information.

CT - *Computed Tomography*, a medical and industrial imaging technique that involves scanning a patient or an object using X-rays and reconstructing projection images to form a 3D volumetric dataset that can be subsequently displayed using a number of different approaches such as volume rendering. See section 2.1 for further information.

CUDA - *Compute Unified Device Architecture* - a GPGPU programming language designed by NVIDIA for their series of consumer and professional-grade graphics cards. See section 2.3.1 for further information.

Energy bin - see **Spectral CT dataset**.

GPGPU - *General Purpose Computing on Graphics Processing Units* - a modern computing technique that involves using one or more GPUs as co-processors to perform general purpose (that is, not necessarily graphics-related) work. Easily parallelisable tasks such as matrix multiplication, fast Fourier transform calculation or n-body simulation are good examples of problems where GPGPU can be used effectively. See section 2.3 for further information.

GPU - *Graphics Processing Unit*, a unit of computer hardware that is specially designed for massively parallel processing of vector and scalar data for the purpose of rendering 2D or 3D scenes. GPU design is based on principles different to

those of a CPU: GPUs contain large numbers of small, relatively slow cores and emphasise parallel computation, whereas CPUs contain few cores, but achieve excellent single-threaded performance due to advanced instruction pipelines, branch prediction hardware and fast caches. See section 2.3 for further information.

MARS project - *Medipix All-Resoltion System* - a collaboration between the University of Canterbury, University of Otago and other partners worldwide. The aim of the project is to create a complete spectral CT system and introduce it into clinical use. See section 2.1.1 for further information.

Octree - a spatial data structure generated by subdividing a three-dimensional dataset into blocks and where each node has 0 or 8 children. In volume rendering algorithms, octrees can be used for empty space skipping or adaptive sampling. See section 4.4.4 for further information.

Tomographic reconstruction - the process of transforming projection images acquired by a CT scanner into cross-sectional slices. Techniques such as filtered back projection and algebraic reconstruction have been developed for this purpose. See section 2.1 for further information.

Slice (as applied to volume data) - a single section through a volumetric dataset. A volume dataset may be stored as a series of slices, each one encoded as an image (for instance, in a TIFF or DICOM format).

Spectral CT - *Spectral Computed Tomography* - a novel medical imaging technique that involves simultaneously measuring X-ray attenuation at several different energy levels to create a number of datasets from a single scan. See section 2.1 for further information.

Spectral CT dataset - a 4D dataset (consisting of a set of co-registered 3D datasets) generated by reconstructing projection data from a spectral CT scanner. Each 3D dataset (also referred to as an *energy bin*) describes the attenuation of X-rays over a certain energy range, as measured by the detector.

Vector quantisation - an asymmetric compression technique that involves reducing an input vector to a single index into a codebook. The codebook is small (generally no larger than 1000-16000 entries) in comparison to the total size of the volume (potentially millions of vectors). Decompression is then performed by

retrieving vectors from the codebook using stored index values. See section 2.4.2 for further information.

Volume dataset (also known as a *volumetric dataset* or a *volume*) - a dataset with its elements (voxels) located on a regular grid in 3D space.

Volume rendering - a visualisation technique that allows the generation of high-quality images from voxel-based datasets, such as those obtained by reconstructing CT data. Volume rendering approaches are based on ray casting, that is, sending rays through the volume dataset that slowly accumulate colour and opacity by sampling inside it. See section 2.2.2.1 for further information.

Voxel - *volumetric pixel* - an element of a volumetric dataset that represents a single value on a regular grid in 3D space. Special techniques such as splatting or volume rendering must be used to visualise datasets composed of voxels.